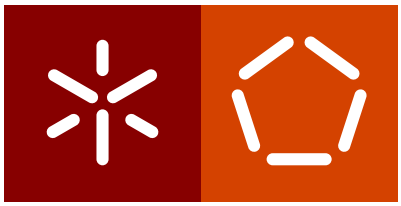
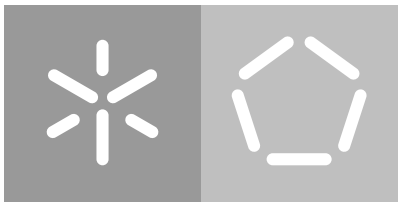


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Esquível Costa

Integration of Post-Quantum Cryptography in the TLS Protocol (LWE Option)

October 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Esquível Costa

Integration of Post-Quantum Cryptography in the TLS Protocol (LWE Option)

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
José Manuel E. Valença

October 2019

ACKNOWLEDGEMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project: UID/EEA/50014/2019.

The work conducted during this thesis could not have been successfully accomplished without the unwavering and unconditional support from my thesis supervisor, José Manuel Valença, whose expertise on the subject, patience and willingness to help with every obstacle encountered along the way proved invaluable to accomplish the objectives proposed. The door was never closed whenever a problem that I needed help with came up, and in multiple occasions he provided me with additional material and ideas that helped shape this work from the very beginning. I could not be more grateful for all the help and support professor Valença gave me throughout this thesis.

I would also like to thank my family, who gave continuous support throughout the entire development of this thesis, as well as throughout the entire journey that began in my first year in the university. Specifically, I express my profound gratitude to my parents, who undoubtedly helped me stay motivated and determined to continue moving forward whenever obstacles appeared in my way. I would also like to extend my gratitude to my girlfriend, who was always present and ready to help however possible, and who provided valuable insight on multiple occasions during the writing of this thesis. Without their support, this work would not have been possible.

ABSTRACT

With the possibility of quantum computers making an appearance, possibly capable of breaking several well established and widespread crytposystems (especially those that implement public key cryptography), necessity has arisen to create new cryptographic algorithms which remain safe even against adversaries using quantum computers.

Several algorithms based on different mathematical problems have been proposed which are considered to be hard to solve with quantum computers. In recent years, a new lattice-based mathematical problem called Learning With Errors (and its variant Ring - Learning With Errors) was introduced, and several cryptosystems based on this problem were introduced, some of which are becoming practical enough to compete with traditional schemes that have been used for decades.

The primary focus in this work is the implementation of two Ring - Learning With Errors based schemes (one key exchange mechanism and one digital signature scheme) on the TLS protocol via the OpenSSL library as a way of checking their overall viability in real-world scenarios, by comparing them to classical schemes implementing the same functionalities.

RESUMO

Com a possibilidade do surgimento dos primeiros computadores quânticos, possivelmente capazes de quebrar muitos dos cripto-sistemas bem difundidos e considerados seguros, tornou-se necessário tomar precauções com a criação de novas técnicas criptográficas que visam manter as suas propriedades de segurança mesmo contra adversários que usem computadores quânticos.

Existem já muitas propostas de algoritmos baseados em problemas matemáticos distintos que são considerados difíceis de resolver recorrendo a computadores quânticos. Recentemente, foi introduzido um novo problema baseado em reticulados denominado de Learning With Errors (e a sua variante Ring - Learning With Errors), e consequentemente foram propostos vários cripto-sistemas baseados nesse problema, alguns dos quais começam já a ser utilizáveis ao ponto de poderem ser comparados com os esquemas clássicos usados há décadas.

O foco principal neste trabalho é a implementação de dois esquemas baseados no problema Ring - Learning With Errors (mais precisamente, um esquema de troca de chaves e uma assinatura digital) no protocolo TLS através da sua integração no OpenSSL como forma de verificar a sua viabilidade em contextos reais, comparando-os com esquemas clássicos que implementem as mesmas funcionalidades.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

The security of most public key cryptosystems used today is based on problems such as the integer factorization and discrete logarithm. There is currently no known algorithm capable of running on a classical computer that can solve these problems in polynomial time, although there is no formal proof that such an algorithm doesn't exist. Both problems are, however, solvable in polynomial time using a quantum algorithm known as Shor's algorithm [Sho94]. By being capable of solving these problems, even for a set of large parameters, otherwise considered safe by today's standard, a quantum computer running Shor's algorithm could recover private information from public one, which renders a public key cryptosystem unsafe. Since most online services nowadays use asymmetric cryptography as means of authentication, as well as for being capable of encrypting traffic between two entities through the usage of key exchange mechanisms, a breakthrough in quantum computing that results in the appearance of quantum computers powerful enough to run Shor's algorithm could mean that most services on the internet would have their communications compromised on a massive scale, rendering internet browsing unsafe worldwide.

Lattice based cryptography has been on the rise as a promising alternative to classical asymmetric cryptographic algorithms, which unlike the latter is based on hard problems that resist even quantum computers. By having a fundamentally different hard problem than those of classic number-theoretic primitives, lattice based cryptography resists Shor's algorithm, and so far there is no known algorithm, both quantum or classical, that can solve hard lattice problems in polynomial time. However, increased security isn't the only benefit lattice cryptography provides. In fact, a number of cryptographic functionalities were made possible, such as fully-homomorphic encryption, that weren't achievable previously by classical cryptography.

In an effort to prepare for the eventual rise of quantum computers, the National Institute of Standards and Technology (NIST) started a new process which, for the next few years, aims at selecting one or more algorithms that are quantum-resistant and standardize them for widespread use. The process started in December of 2016, and a call for proposals was issued, with a deadline for the first round of submissions on November 30, 2017. The round 2 candidates were announced on January 30, 2019. From the 23 digital signature schemes

submitted for round 1, only 9 managed to pass to the second round. Likewise, from the 59 public key encryption and key establishment algorithms submitted in the first round, only 17 made it to round 2.

The analysis and standardization of public key cryptography that is quantum-resistant is extremely important. According to some predictions, quantum computers capable of breaking modern public key cryptography could become a reality in more or less 20 years and, according to NIST [ST17], the public key infrastructure currently in use throughout the world also took around 20 years to be deployed. Therefore, the soonest quantum-resistant schemes can be standardized, the soonest they can be deployed, which decreases the chances of having most modern public key cryptography broken and worldwide communications compromised.

The Learning With Errors (LWE) problem is a problem introduced by Oded Regev in [Reg05] that can be reduced in complexity to worst-case lattice problems, which are known to be hard to solve. Although there have been some cryptosystems based on this problem as is, the most recent and promising cryptosystems are related to a variant of the former called Ring - Learning With Errors (R-LWE) or Learning With Errors Over Rings, introduced in [LPR10] by Lyubashevsky, Peikert, and Regev. This problem is shown by the authors to have similar security properties to the LWE problem, while at the same time being more efficient by requiring parameters with smaller sizes.

Throughout the years since its introduction, the R-LWE served as a base for several cryptographic schemes, including not only several signature algorithms and key exchange mechanisms, but also fully homomorphic encryption schemes. A more detailed overview on some digital signatures and key exchange mechanisms will be given in chapter 2.

Safe communications on the web are usually accomplished using the TLS protocol, a successor of the now deprecated SSL protocol. In essence, this protocol works on top of a transport protocol like TCP and provides four different cryptographic applications for safe communications, those being symmetric encryption, key exchange mechanisms, authentication mechanisms (digital signatures) and message authentication codes. Since the main purpose of post-quantum cryptography is to provide new asymmetric public key cryptosystems that resist quantum computers, the attention in this paper is focused on digital signatures and key exchange mechanisms, both of which are vulnerable to algorithms solving integer factorization and discrete logarithms.

Key exchange mechanisms currently employed in the TLS protocol are used to determine a handshake between two communicating entities, usually a server and a client, by generating a secret key to be used through the session to symmetrically encrypt messages sent between the two participating entities. Since these secret keys are unique per session, the handshake protocol has to occur every time a new session is created.

Digital signatures on TLS provide authentication between parties through the use of digital certificates. When a client starts communicating with a server, it requests the server's certificate, which contains its public signing key, to ensure the server is in fact a trusted entity. The client can use the server public key to verify signatures made by said server. The client can also be requested to provide a certificate to authenticate itself, in which case the process is similar, where both entities request the other's certificate, validate them and use the public keys in said certificates to validate each others signatures.

The TLS protocol uses the algorithms for each cryptographic functionality in the form of ciphersuites. Each ciphersuite is represented by a string that specifies which algorithm is to be used for every cryptographic functionality. Picture 1 shows an example of a ciphersuite currently used in TLS by the Microsoft Windows 10 operating system.

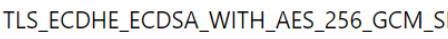


Figure 1: TLS ciphersuite example

In order to use new algorithms in this protocol, it is necessary to add new ciphersuites that include the new cryptographic functions that the involved entities can now use in the communication. For instance, if both the client and server have access to a quantum-safe key exchange mechanism, then it can be used to create the shared secret key for symmetric encryption in that session as would have been done with a classical key exchange mechanism like the ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) shown in 1.

The currently best established implementation of the TLS protocol is arguably present in the OpenSSL toolkit, which by being open-source facilitates the possible inclusion of new algorithms, since all the code is accessible to anyone that might want to read and perhaps even contribute. Because of its widespread use, allied to the accessibility of the source code, integrating new digital signature and key exchange schemes in OpenSSL is a good way to include these algorithms in the TLS protocol. It is noteworthy, however, that the source code implementation is highly complex, containing a lot of files that need to be altered and possibly thousands of lines of code that need to be added or modified [SM16].

Given the highly complex implementation, which has suffered a lot of changes throughout the years, rendering tutorials of previous versions less useful, the integration can be achieved through different means. One such way consists of trying to follow one or more examples of previous successful attempts at integrating new algorithms. Even though examples like that exist, those implementations occurred in previous versions of OpenSSL, which was significantly modified in certain parts with subsequent versions. Another way of including a new algorithm could be through the use of engines. An engine is essentially a way of running external implementations of an algorithm on OpenSSL. However, there is a way which is probably the most useful one out of these three, which is making use of the Open Quantum Safe Project , which will be detailed in chapter 4. Essentially, it's a

project introduced in [SM16] which offers a much simpler and more modular way of adding post-quantum algorithms to OpenSSL, by making use of an external library to which the algorithms must be added[1].

Even though the primary objective of this dissertation is to integrate post-quantum, "learning with errors" based algorithms on TLS, there are great benefits in trying to extend those algorithms to different tools that also benefit from widespread usage across the world. One such tool is the **cryptography** package from the programming language Python, which is a package that offers a rich set of different cryptographic functionalities, among which are several digital signatures and key agreement mechanisms. Because it is heavily used worldwide, and currently does not support post-quantum algorithms, it would be interesting to integrate the Open Quantum Safe algorithms in this package, in order to be able to test their standalone usage in a simpler, more accessible way.

Since integration on OpenSSL was accomplished using the Open Quantum Safe Project by adding the new algorithms to the project's library, integration on Python's **cryptography** was accomplished by integrating the whole Open Quantum Safe library in the package, allowing the usage of every algorithm supported in the project directly on Python, instead of only the new ones added in the context of this dissertation.

The next chapters introduce some of the most well known and influential contributions in post-quantum cryptography, starting on the origin of the "learning with errors" problem, since it is the base problem on which the proposed cryptosystems stand, and ending on some of the most recent systems proposed, which are perhaps the systems with the best chances of becoming standards in the future. Because this work revolves around key agreement schemes as well as digital signatures, greater focus is given when analyzing literature related to these mechanisms.

Chapter 2 is divided into three sections, namely sections 2.1 and 2.2, which introduce a revision of the current state of the art, by describing the most important contributions on this subject, and section 2.3, which refers to similar work accomplished in recent years.

Chapter 3 introduces some mathematical notations and some cryptographic schemes in more detail.

Chapters 4 and 5 delve into the algorithm's integration details, divided into two parts: integration on the Open Quantum Safe Project liboqs library, which includes a section regarding the **cryptography** integration, and integration into OpenSSL using the liboqs library.

Chapter 6 showcases some tests regarding different algorithms (classical, post-quantum and hybrid variants), both in the generation of digital certificates and in the establishment of TLS connections.

1 The project itself isn't entirely made for OpenSSL, nor is it dependant on it. Instead, it is a standalone C library implementing post-quantum algorithms. The authors then integrated the library on a fork of OpenSSL, making the process of including new algorithms more modular.

Finally, some reflections on the obtained results, as well as future works suggestions, can be seen in chapter 7.

2

BACKGROUND LITERATURE

Since the award winning work of Oded Regev in [Reg05], which introduced the learning with errors (LWE) problem, numerous cryptographic functionalities based on this problem have been proposed by a myriad of different authors. Because LWE cryptography is relatively recent when compared to classical cryptography, one problem that arose throughout the years is finding a concrete instantiation of parameters to be used when using LWE cryptography.

Even though public key cryptography has received more attention in the post-quantum scenario, it doesn't mean that symmetric cryptography is safe from attacks from quantum computers, as pointed in [DW17] and [DDW18][1]. In fact, the authors devised the first quantum key-recovery attack on Feistel block ciphers, which is the base for multiple block cipher schemes used today. Even though these attacks do not use Shor's algorithm, since they don't explore the problems described above, they do make use of quantum computers, meaning symmetric encryption is also a subject of interest in post-quantum cryptography that shouldn't be overlooked.

2.1 LWE BASED SCHEMES

Several cryptographic schemes based on the LWE problem have been proposed since its initial introduction in 2005 [Reg05]. These schemes are not the most common nowadays, mainly because the ring variant of this problem also benefits from strong security assumptions [LPR10], while at the same time allowing for shorter parameters overall, which leads to the development of more practical schemes. Nevertheless, it is relevant to briefly review some LWE based schemes if not for anything else because it presents an overview of what has been proposed throughout the years, and why the R-LWE variant made significant improvements to the initial problem. Furthermore, there is a key exchange mechanism, called Frodo, that was submitted to NIST and passed to the second round along with other R-LWE based schemes, which shows that schemes based on this problem may still be viable options for standardization.

1 The authors also mention Grover's algorithm [Gro96], a quantum algorithm that reduces search complexity for a given dimension N parameters to approximately $\sqrt{N}$.

2.1.1 *Encryption Schemes*

The first LWE based encryption scheme was proposed by Oded Regev in [Reg05]. This public key encryption scheme, as is the case with every subsequent scheme based on this problem, works under the assumption that the LWE problem is as hard to solve as worst-case lattice problems like the Shortest Vector Problem (SVP), as is proved by the author in the same paper. As virtually all public key encryption schemes, whether classical or quantum-safe, this scheme is composed of 3 main functions, namely private and public key generation, encryption and decryption.

According to Regev, the set of parameters proposed in the paper for this encryption scheme can guaranty the security of the scheme. For a given security parameter n, with the proposed parameter set, the public key size is $\tilde{O}(n^2)$. However, this scheme was still much more efficient than previous public key lattice-based cryptosystems, since those required public keys of size $\tilde{O}(n^4)$. Moreover, encryption of a message would see its size grow by $\tilde{O}(n^2)$, while the author's proposed scheme would grow a message by $\tilde{O}(n)$.

The author furthers matters even more by proposing a method of reducing the size of the public key to $\tilde{O}(n)$. That solution would, however, involve the need for every user of the cryptosystem to share a fixed random set of vectors a_1 to a_m, where m is a parameter of the scheme, instead of generating said vectors in the generation of the public key, further reducing its size. According to the author, this solution does not affect the security of the cryptosystem in any way.

Even though the scheme proposed is far more efficient than previous schemes, it still involves the use of public keys that can grow to impractical sizes, especially in situations where large parameters might be necessary.

Other encryption schemes worthy of notice are that which was proposed by Chris Peikert in [Pei08], which was proven by the author to be safe against chosen plaintext attacks, or another cryptosystem proposed by Miklós Ajtai and Cynthia Dwork in [AD97] which precedes both of the schemes aforementioned and which, while not being based on the learning with errors problem, is based on a special variant of the SVP problem, called unique-SVP. This scheme is one of the schemes mentioned above which is significantly less efficient than that proposed by Regev.

Lindner and Peikert introduced in [LP10] an improved cryptosystem which the authors characterize as being more compact and efficient than other cryptosystems proposed at the time, such as the one introduced by Regev, while at the same time providing stronger security levels. It is noteworthy that even though the proposed system requires lower key sizes by an order of 10 compared to previous models, the authors recognize that the keys are still too large to be used in most applications, pointing to the R-LWE as a way to reduce parameter sizes significantly.

2.1.2 *Key Exchange Schemes*

Key exchange (or key agreement) mechanisms are used to determine a shared key between parties over an untrusted network without revealing any private information that could render the whole communication unsafe. However, there exist multiple variants that can be used to achieve this purpose (for instance, authenticated and unauthenticated schemes). Since key agreement schemes used in the TLS protocol are mostly unauthenticated, like Diffie-Hellman (although there are exceptions, like RSA for key agreement, which can authenticate the server), it makes sense that an attempt to integrate a new key agreement based on lattice cryptography would share the same property, mainly because the general structure of the protocol would remain the same, maintaining a similar interface to that of classical algorithms.

Most LWE and R-LWE schemes to be introduced are essentially key encapsulation mechanisms (KEM), although some are defined as key exchange schemes and are structurally compared to the Diffie-Hellman key exchange. Following the definition Chris Peikert provides in [Pei14], a KEM (or key transport mechanism) is a set of four algorithms (Setup, Gen, Encaps, Decaps), where:

- *Setup* produces a public parameter;
- *Gen* outputs a public (encapsulation) key pk and a private (decapsulation) key sk, given the public parameter;
- *Encaps* uses the public key and parameter and generates a ciphertext $c \in C$ and a key $k \in K$;
- *Decaps* uses the private key and ciphertext to produce a $k \in K$.

Ding, Xie, and Lin [DXL12] define one of the drawbacks of a key transport mechanism (Peikert defines key transport mechanism as a synonym of key encapsulation mechanism), which is the fact that the secret shared between the two parties will be determined exclusively by one of them. However, some authors, like Alkim et al. [Alk+16], present two variants of a scheme, one that derives the shared secret from both parties involved in the exchange and one that requires only one of them to derive it. Both schemes are presented with KEM notation (i.e., with *Gen, Encaps and Decaps*), yet the authors seem to use the terms KEM and key exchange as synonyms, deviating from the definitions in [DXL12]. In [Bos+14], the authors define the scheme as a key exchange scheme, and yet the same scheme is mentioned in [Alk+15] with KEM notation. Finally, in [Bos+17], the authors use the KEM proposed to create a key exchange protocol, which is obtained as a "direct application of the key encapsulation mechanism" [Bos+17].

It seems reasonable to conclude that, even though some of the literature regarding LWE and R-LWE schemes isn't completely clear in respect to how exactly each scheme should

be called, all of them seem to be have something in common, which is that all schemes implement a protocol to derive a shared secret between two peers on an untrusted channel. It seems to be the case that the term key exchange is adequately described by this definition, and so the proposed schemes introduced in this chapter will use this general definition.

In [DXL12] the authors propose a new key exchange protocol similar to the Diffie-Hellman exchange, but based on LWE for hardness instead of the discrete logarithm problem. As is pointed out by the authors, one cryptographic property of great relevance in a cryptographic algorithm, in this case a key exchange mechanism, is forward security, which the Diffie-Hellman protocol possesses, contrarily to encryption based key exchange mechanisms (RSA for example). When establishing a secure connection between two parties using the Diffie-Hellman protocol, each session key derived from the algorithm is ephemeral, which means that even if an adversary can recover the session key and decrypt messages, other messages exchanged between parties in previous sessions are still protected, since each session has a different key unrelated to any other used before or after. Exchanging keys using an encryption scheme like RSA means that if somehow one private key used for key exchange would be compromised, every message in every session between parties with a compromised private key could be recovered, because the agreed-upon key to encrypt every message would be decrypted using the private key.

The scheme is based on a notion present in the traditional Diffie-Hellman scheme, which is the commutativity of exponentiation, i.e., $g^{xy} = (g^x)^y = (g^y)^x$. This property is used in the scheme via a multiplication of two vectors with a square matrix M, i.e., $x^T My = (x^T M)y = x^T(My)$.

The shared key agreed by the parties results from the use of a function called by the authors as a robust extractor, which essentially outputs the same value from two different inputs if those inputs are close enough to each other (the difference between the two values is upper-bounded by a certain value δ) and if their difference is an even number. Provided the two conditions are met, the authors provide a mathematical proof that the extractor always outputs identical values, meaning both parties can derive the exact same session key from approximate ones.

One interesting aspect of the key agreement proposed, besides the use of the extractor aforementioned, is that contrarily to most systems that sample a secret from $\mathbb{Z}_q^n$, this system samples the secrets from the error distribution, basing itself on a variant of the LWE problem called the Hermit Normal Form LWE (HNF-LWE), which doesn't affect the hardness of the problem [App+09].

In a nutshell, after generating parameters q, n and α, as well as uniformly random matrix M in $\mathbb{Z}_q^{n\times n}$, the key exchange protocol involves Alice and Bob each choosing a secret vector s_A or s_B from a Gaussian distribution $\mathcal{D}_{\mathbb{Z}^n,\alpha q}$, an error vector e_A or e_B from the same distribution and computing, respectively, $p_A = Ms_A + 2e_A$ and $p_B = M^T s_B + 2e_B$, which

represent the public key to be sent to the corresponding party. Internally, each party then respectively calculates values $K_A = s_A^T p_B + 2e'_A$ and $K_B = p_A^T s_B + 2e'_B$ using the key sent by the other party. These values are close to each other due to the fact that both the secret and error vectors have small Euclidean norms compared to q. Alice then derives the shared key by applying the robust extractor function to K_A, and Bob by doing the same to K_B, which is guaranteed to provide an identical shared key if the protocol is followed correctly. The public and shared keys are all calculated $\mod q$.

The proposed scheme was considered by the authors as being more efficient than other key exchange schemes based on LWE encryption. Still, the authors also propose a variant of the key exchange described based on the R-LWE problem, further improving the performance of the scheme. Also, based on the R-LWE variant scheme, they extend the protocol to support multiple parties at the same time, but leave its mathematical proof as an open problem.

The scheme described inspired other, more recent proposals, namely the scheme proposed in [Bos+14] and, consequently, [Alk+15], which will be described in section 2.2.

In [KV09], the authors describe the first password-based authenticated key exchange (PAKE) scheme with security based on lattices. Essentially, this is a protocol that allows two peers to derive a shared symmetric key from a shared, usually low entropy password. The proposed scheme is based on an encryption scheme which in turn is based on the LWE problem, and is proven by the authors to be IND-CCA secure. In [KV10] the same authors introduce a framework to create a generic one-round PAKE algorithm, in which parties send only one message to each other and do it simultaneously.

In 2016, the authors in [Bos+16] introduced a new scheme called Frodo and demonstrated that, even though it is based on the LWE problem, it can still be efficient enough to be used in real-world scenarios. Even more, the authors compare the Frodo scheme with two R-LWE schemes (BCNS [Bos+14] and NewHope [Alk+15]) and conclude that, while it is true that schemes that are based on the R-LWE problem do tend to have fewer message sizes and latency on communications, the differences are small enough to still make the scheme a viable option for replacing the usage of a classical scheme.

It is worth mentioning that when testing the standalone implementation of the algorithm, the scheme revealed to be 1.8 times slower than the classical ECDH **nistp256** and 9 times slower than NewHope, which is based on R-LWE. The authors point out, however, that when using a protocol like TLS, the gap between Frodo and other schemes becomes much narrower. Because of all the different components involved in the TLS protocol, and because it can possess several bottlenecks that aren't directly related to the key exchange algorithm itself, differences in loading times are severely reduced, effectively showing that the standalone implementations do not dictate, by themselves, how viable the scheme can be.

The slightly increased loading times and message sizes, as explained by the authors, when compared to the loading speed of an average web page, does not greatly impact

the experience. While the values are worse than those that can be obtained by ECDH or NewHope, in most cases the differences are small enough as to not even be perceivable.

The increase in efficiency can be attributed to two major differences when compared to other LWE schemes: a better alternative for noise sampling, i.e., the usage of noise distributions that are more efficient to sample than the discrete Gaussian distributions and the optimization of a pseudo-random generator for public parameters, namely, the public matrix $\mathbf{A}$ to be used by both parties in a communication. Furthermore, since the matrix is generated from a small seed, it allows for smaller communication sizes since there is no need anymore to transmit the whole matrix, sufficing it instead to transmit the small seed. As is pointed out by the authors, the optimization of this pseudo-random generator was important to achieve the high efficiency needed to compete against R-LWE schemes, since it still is computationally expensive, with around 40% of the whole computation time for each peer.

Since the Frodo scheme is based on the LWE problem, the authors explain why they didn't base it instead on the more efficient R-LWE variant. Essentially, even though the R-LWE variant is currently as hard to solve as the original LWE problem, for the same parameter size, it remains unknown if one day it might be proven to be much easier to break than the original LWE problem, since the problem is based on the worst-case hardness of problems in ideal lattices, as opposed to generic lattices. It is because of the possibility that one day there might be a new discovery that renders worst-case problems in ideal lattices much easier to solve as opposed to generic lattices that the authors seem to believe there is a clear benefit in designing a scheme that is viable in terms of real-world usage that is simply based on the more generic and well studied LWE problem.

2.1.3 *Signature Schemes*

Until the works of Gentry, Peikert, and Vaikuntanathan [GPV07] and Lyubashevsky and Micciancio [LM08] there had been no lattice-based signature scheme to be proposed with a formal security proof, although some schemes had been proposed as far back as 1997 in [GGH97], as well as in 2001 in [HPS01]. These schemes did not provide any formal security proof, and where eventually broken [Duc+13; Lyu11].

Both schemes proposed have security proofs. The scheme in [GPV07] uses a hash-and-sign technique to sign messages, which results in very large signatures that can be impractical in many real-world use scenarios. The scheme proposed in [LM08], while possessing smaller signature sizes, is a one-time signature scheme, which as is mentioned in [GPV07], [Lyu09] and [Lyu11], would need hash or Merkle trees to become a full signature scheme, which might not be the best solution in practice. As is pointed out by the authors, not only is this

scheme capable of producing shorter signatures and keys, but is also efficient signing and verifying messages, with time complexity being "almost linear" [LM08].

One signature scheme proposal by Lyubashevsky in [Lyu09] was introduced in 2009 which employed a framework called Fiat-Shamir [FS87], traditionally used for number-theoretic schemes. The author demonstrates in this paper that the framework can be used in lattice based cryptography by creating a signature scheme capable of creating signatures much smaller than those of previous aforementioned schemes. As is stated by the author, the signatures produced have around 50000 bits, which is a major improvement in size.

Even though the signature size in [Lyu09] is significantly lower than that of other schemes (some have signatures with millions of bits in size), as is stated by the same author in a different, posterior paper [Lyu11], "it is still not as small as one would like". The author proposes a new scheme, provably secure in the random oracle model, that greatly reduces key and signature sizes compared to hash-and-signature schemes proposed before. Furthermore, it is considered by the author as being relatively simple, and capable of having more efficient signatures by changing the base of the hardness to the LWE problem instead of other lattice-based problems. Finally, the author provides, as is occasionally done, a variant of the scheme based on the R-LWE problem, reducing even more key sizes and computational cost to create and verify signatures.

Most signature schemes based on hard problems on lattices, especially the LWE, suffer from some form of impracticality caused by key sizes necessary to create provably secure systems, as well as from the complexity of operations like signing and verifying. As was the case with key exchange mechanisms aforementioned, some authors, in their proposals, include a scheme variant where the hardness is based on the R-LWE as perhaps the best way to increase efficiency without sacrificing security, making schemes much more practical for real-world usage.

2.2 R-LWE BASED SCHEMES

Modern lattice cryptography has seen a number of proposed schemes based on the R-LWE problem, which since its introduction by Lyubashevsky, Peikert, and Regev in [LPR10], has been shown to allow for constructions that benefit from similar security problems, while needing much smaller key sizes and performing more efficiently.

As is stated in [LPR10], there are multiple reasons for the increased efficiency of R-LWE when compared to the traditional LWE. One boost in efficiency comes from the most complex operations in R-LWE schemes usually being polynomial multiplications, which can be accomplished with $O(n \log n)$ complexity using highly efficient algorithms like the NTT (number theoretic transform) or simply using FFT (fast fourier transforms) to multiply large polynomials (with large degree n). This accomplishes polynomial multiplication significantly

faster than the traditional multiplication algorithms, with complexity $O(n^2)$. On top of that, while in the LWE problem each product between vectors a and s generates a single number, the same product in the ring variant generates n values in $\mathbb{Z}_q$ and, as mentioned above, the cost of the $a \times s$ is reduced via efficient polynomial multiplication algorithms already well established and with several implementations. Furthermore, samples in the ring variant can be reduced by a factor of n when compared to the LWE problem, reducing key sizes, which was, to some extent, a problem in every proposed scheme based on the LWE problem.

Because of such an improvement in efficiency, creating schemes that can effectively "replace" number-theoretic cryptosystems became more feasible, leading to proposals that pose as some of the currently best schemes in post-quantum cryptography, and possibly suitable for the integration on the TLS protocol. Among other properties, some that are especially useful in such a scheme are as follows:

- Efficiency: the algorithm should be efficient enough to, at least, not cause significant delay compared to other classical algorithms in real-life scenarios;
- Security: the scheme must be as secure as possible, so as to not lose any value regarding safety when compared to already well established number-theoretic algorithms;
- Simplicity: mainly in the current context, it's useful to take into account complexity of implementation, since the integration of an algorithm into OpenSSL is already a fairly complex process, involving potentially hundreds or even thousands of changes in the source code [JN16]. Even when integrating via Open Quantum Safe, it still is useful to select algorithms that are somewhat compatible to the project's API.

There is also another property that is important to consider, which is the similarity a scheme bears to previous classical schemes that are already implemented on OpenSSL and TLS [Bos+14; Alk+15]. This property makes particular sense in this context as it is greatly beneficial to be able to simply "substitute" an existing, already integrated algorithm on OpenSSL as a means to integrate post-quantum cryptographic schemes.

Taking the properties aforementioned in careful consideration, the background work revision will be focused on proposed schemes for both signature schemes and key exchange mechanisms that share these properties to some extent.

2.2.1 *Key Exchange Schemes*

Perhaps one of the most relevant R-LWE key exchange mechanism to mention is that proposed by Ding, Xie, and Lin in [DXL12], which was already mentioned in the LWE variant section. This scheme, as was previously mentioned, is phrased as a Diffie-Hellman like scheme. Furthermore, it has been improved by Peikert [Pei14] regarding the reconciliation

mechanism (i.e., the mechanism that assures Alice and Bob share an identical key based on an approximate one), and served as a base for a new R-LWE key exchange protocol in [Bos+14], which was then integrated on OpenSSL. The scheme was then revised and some alterations were made to increase performance and security. The resulting scheme, called NewHope [Alk+15], was then further simplified for easier implementation by the same authors in [Alk+16]. This new variant avoids the reconciliation mechanisms used by previous schemes, which according to the authors has a relatively low impact on the overall performance, and no impact whatsoever on the security of the NewHope scheme.

The R-LWE variant in [DXL12] is a fairly straightforward change from the initial LWE variant, still making use of the robust extractor as a reconciliation mechanism for deriving identical keys. Because of the properties of the R-LWE variant, this scheme is capable of functioning with fewer parameter sizes and operates with lower computational complexity, which is estimated by the number of polynomial multiplications in the scheme. In the comparison made by the authors in [DXL12] against proposed schemes in [LPR10] and [SS11], they reveal that the scheme has shorter parameters as well as lower communication and computational complexity.

[Pei14] introduces in his paper a new reconciliation mechanism that is fundamentally different from that proposed in [DXL12], and one that is actually general enough to be used in every key exchange mechanism, while effectively reducing the ciphertext to be sent in size approximately in half.

Essentially, the mechanism uses a cross-rounding technique to make sure every bit derived from two "approximate" shared keys is identical while remaining unbiased. The mechanism is first illustrated using a parameter q which, at first, is considered to be even. When q is even, the cross-rounding function is as follows (the equation is present in [Pei14]):

$$\langle v \rangle_2 := \lfloor \frac{4}{q}.v \rfloor mod 2 \tag{1}$$

As the author points out, as long as q is greater than 2 and even, if $v \in \mathbb{Z}_q$ is uniformly random, then $\langle v \rangle_2$ is also uniformly random. If, however, q is odd, then $\langle v \rangle$ is biased towards 0 [Pei14].

As is the case with nearly every scheme using rings, q is usually a big prime, therefore being odd. As is, the cross-rounding would create bias in the determined bit. To solve the problem, the author introduces a tweak which involves temporarily working modulo $2q$, by defining a new function *dbl* that outputs a value $\bar{v} = 2v - \bar{e} \in \mathbb{Z}_{2q}$, for $\bar{e} \in \{0, 1\}$ uniformly sampled or $\bar{e} \in \{-1, 0, 1\}$ with probability $\frac{1}{2}$ for 0 and $\frac{1}{4}$ for 1 and -1.

The reconciliation mechanism works via a function *rec* which receives a peer public key and a mask $\langle v \rangle_2$ which is a string of bits derived from the peer's approximate shared key v. Peikert shows (on claim **3.1** and claim **3.3**, section **3** of [Pei14]) that, while $\lfloor v \rceil_2$, which is the shared key, is a private parameter, it is safe to send $\langle v \rangle_2$ as a public parameter, since it

perfectly hides $\lfloor v \rceil_2$. After receiving the peer's public key and the approximate shared key mask $\langle v \rangle_2$, the other peer can then derive the exact agreed upon key $\lfloor v \rceil_2$ by applying the *rec* function using as arguments both the public key and the mask. Claim **3.2** holds that for peer approximate keys w and v, and for small error value $e \in [-\frac{q}{8}, \frac{q}{8}]$, if $w = v + e$, i.e., if the approximate keys are "close" to each other, then $rec(w, \langle v \rangle_2) = \lfloor v \rceil_2$. Using this mechanism both peers can agree on a private, exact shared key $\lfloor v \rceil_2$ exchanging only public information, as is the objective of a key exchange scheme.

The reconciliation mechanism is presented in [Pei14] as being used on a value $v \in \mathbb{Z}_q$. To apply it to polynomials in R-LWE schemes, the mechanism must be applied to each coordinate (polynomial coefficient) independently. The process is identical to that which was described before.

In [Bos+14], the authors implement an unauthenticated key exchange mechanism by using a key exchange algorithm similar to that of [DXL12], but with the reconciliation mechanism described in [Pei14], given its increased performance. The scheme proposed is phrased as a key exchange scheme and not a KEM, to allow for an easier comparison to Diffie-Hellman like schemes. Nevertheless, it can be interpreted as a KEM, with the usual *Setup*, *Gen*, *Encaps* and *Decaps* methods, in which the *Encaps* function outputs a ciphertext c and a key to be used for symmetric encryption (the agreed-upon key), and the *Decaps* method receives the peer's public key and uses it to recover the same symmetric encryption key from the ciphertext received.

The authors go through great lengths to make sure the scheme is safe both against classical and quantum attacks, and take a rather conservative approach by choosing the degree n in the polynomial ring to be 1024, as opposed to the usual 512. Also, two versions of the protocol are implemented, where one version actually runs in constant time, in order to further improve security. In fact, since the algorithm involves sampling from a discrete Gaussian distribution, and because sampling from such a distribution opens the system to side-channel attacks, such as timing attacks [Akl+16a; Cho17; Alk+15], it may be worth the extra effort to reduce chances of a successful attack.

The implementation by Bos et al. [Bos+14] will be further mentioned in 2.3, since the authors then integrate their own implementation on the TLS protocol via OpenSSL.

Following the work in [Bos+14], Alkim et al. [Alk+15] introduce a variant of the unauthenticated scheme with a few changes aiming at increasing both security and efficiency, by proposing new parameters, a new error distribution other than the regular discrete Gaussian and even a new, better reconciliation mechanism. The authors claim to "double the security parameter, halve the communication overhead, and speed up computation by more than a factor of 8".

One major cause for inefficiency pointed out by the authors is the Gaussian sampling, which they point out can be substituted by a centered binomial distribution with minimal

impact in security, while not being as hard to implement. In fact, according to the authors, sampling from a Gaussian distribution only makes a considerable difference when dealing with signature schemes or lattice trapdoors [Alk+15].

As was the case with the implementation in [Bos+14], the authors propose the set of parameters to use in the scheme. While keeping the $n = 1024$, mainly to achieve long-term security, they change the modulus q to 12289, a value much lower than the $2^{32} - 1$ proposed in [Bos+14]. The authors present mainly two reasons for the value, namely the fact that the reconciliation mechanism is more error tolerant, meaning that the error-to-modulus ratio can be improved, leading to better security, and that 12289 is the first prime in which $q \equiv 1 \mod 2n$, which leads to more efficiency when using the number-theoretic transform (NTT).

In relation to the public polynomial a, Alkim et al. provide two reasons to generate a every time a new key exchange occurs: protection against backdoors and against all-for-the-price-of-one attacks. Briefly, a backdoor could be included in the generation of a by an authority with dishonest intents, generating the public polynomial in such a way that could weaken the security of the system.

Supposedly, if an adversary with incredible computational problem would find a way to solve the problem for the polynomial a, even of it took a long time, then every communication using this scheme would be compromised, because every communication used the same a.

By generating a every session, it is possible to avoid these problems by not depending on an authority to generate a and by making the solving of the problem in a long time (like a year) still irrelevant to compromise future communications.

In practical terms, a is generated using an extendable-output function (XOF) officially standardized by NIST in the SHA-3 standard [Fot15] called SHAKE128 (SHAKE128 and SHAKE256 are actually the first XOFs to ever be standardized by NIST). In a nutshell, this function generates a variable length output with 128 bits of security. Even though there is a 256 bit variant, and that it doesn't decrease the scheme's performance significantly, the authors explain that such a variant is not needed since potential collisions are not capable of attacking the system.

The reconciliation mechanism, which allows for a better error tolerance, involves the encoding of 4 polynomial coefficients at a time into a single bit, by determining the bit via the distance of the vector represented by the coefficients to a specific point in a lattice $\tilde{D}_4$ with a basis composed of the three base vectors x, y and z in a $\mathbb{Z}^4$ referential and a fourth vector $g = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. When the key agreement results in an approximate shared key, the client and server have slightly different vectors in the $\tilde{D}_4$ lattice. In essence the difference between the client vector and the closest lattice point is sent to the server which adds it to its own vector. To reduce the amount of information sent, the Voronoi cell defined by the lattice point is subdivided in smaller cells and only information about that particular cell is sent to the server.

However efficient the scheme may be, and it is more efficient than those of Peikert [Pei14] and Ding, Xie, and Lin [DXL12], the reconciliation is, even according to the authors, quite complex, both in the understanding of the technique and the corresponding implementation.

With this in mind, a new variant was proposed by the same authors in [Alk+16], which avoids the whole mechanism described above, at a relatively low trade-off. Specifically, of the two messages exchanged (Alice to Bob and then Bob to Alice), only that from Bob to Alice has an increase in size, and only of 128 additional bytes, not leaving a huge impact on performance.

The new variant, which was named NewHope-Simple, eliminates the reconciliation mechanism by following the "encryption-based approach", instead of the "reconciliation-based approach". In this new variant, Bob generates a uniformly random bit string v to be encoded in a polynomial k. This is accomplished using the function *NHSEncode*, which encodes each bit from v into four coefficients in k, in "steps" of 256. The corresponding function is *NHSDecode*, which takes the four coefficients and calculates the corresponding bit. The next step is to calculate polynomial c (similarly to what was described before) and then calculating $\bar{c}$ using another function called *NHSCompress*, and what it does is basically a modulus switching from q to 8. This step comes from the realization that only the highest bits of c actually matter in the determination of the exact key, while the rest are mainly just noise. By carefully choosing only the highest bits to transmit, the amount of information actually transmitted to Alice is reduced significantly, while still maintaining an acceptable success rate in deriving the exact keys (in fact, the authors mention that they constraint the size reduction only as long as the failure probability, i.e., the probability of different keys being derived, does not exceed that of the original NewHope scheme). When Alice receives the data sent by Bob, she applies the inverse functions (*NHSDecode* and *NHSDecompress*) to obtain the exact shared key.

One important detail regarding both schemes is the use of the number-theoretic transform (NTT) whenever possible to ensure fast polynomial multiplication. Furthermore, the sampling of a can be done already in the NTT domain, increasing performance if needed.

The schemes in [Bos+14], [Alk+15] and [Bos+16] (BCNS, NewHope and Frodo) are secure against passive adversaries, i.e., adversaries that can "see" information exchanged between parties but cannot actively modify it. The authors in [Alk+15] reason that it is more useful to have unauthenticated key exchange schemes and consider advantageous to separate the authentication component from the key exchange into two different algorithms that can be used independently from one another. The authors in [Bos+14] and [Bos+16] integrate their respective algorithms on the TLS protocol and, thus also adopt an unauthenticated approach, closely resembling a Diffie-Hellman like approach.

Another key exchange scheme that was submitted to NIST in 2017, and that also passed the second round, is CRYSTALS-Kyber [Bos+17]. This scheme is based on a generalized

version of the R-LWE problem called Module - Learning With Errors (M-LWE), which will be briefly presented in chapter 3. The authors Bos et al. created a scheme that is IND-CCA2 secure and is constructed from an IND-CPA secure PKE scheme using a variant of the Fujisaki-Okamoto transform [FO99], and therefore is actively secure against adversaries [Pei14].

The authors explain, in an updated version of the paper [Bos+19], why the scheme was designed as IND-CCA2 secure only. Even though some use cases like the TLS protocol do not require that key exchange protocols be actively secure, Bos et al. still believe that the downsides of bringing an IND-CCA2 scheme aren't severe enough to justify "making the scheme less robust".

One interesting aspect of the proposed key exchange is the components used to generate the shared key between the two parties. In the NewHope "encryption-based scheme", for instance, the shared key, which is determined only by one party[2], is derived by hashing the randomly generated value $v \in \{0,1\}^{256}$. In Kyber, the final shared secret is dependant on the public key, which is hashed to compute the first half of the key, and on the ciphertext produced by the encapsulation function, effectively making the secret depend on the two messages exchanged during the protocol (Bob uses Alice's public key and Alice uses Bob's ciphertext).

The approach taken on the "reconciliation-based" version of NewHope [Alk+15] still uses the public key from Alice to generate a shared secret that isn't fully dependent on Bob. However, Bob doesn't generate a random secret to be encapsulated into a ciphertext with this approach.

NewHope and Kyber are based, respectively, on R-LWE and M-LWE. However, there is another scheme called ThreeBears [Ham19], which also passed to the second round, that is based on these two schemes, but with a different ring instantiation. The problem, in this scheme, is a variant of M-LWE called Integer - Module Learning With Errors (I-MLWE), where the ring elements are integers and not polynomials. In the paper introducing the scheme, the author Hamburg also describes what seems to be a general framework for key exchange schemes based on M-LWE , and further specifies the scheme's concrete instantiations. This general framework will be discussed in chapter 3.

2.2.2 *Signatures Schemes*

Following the logic of key exchange mechanisms, signature schemes based on the R-LWE variant come as a sort of solution to the problem of having parameters that are just too high to be practical in most use cases, yet necessary to achieve the necessary levels of security.

2 It is noteworthy, however, that on the "encryption-based" variant [Alk+16], the authors do mention that parts of Alice's public key can be hashed into the final shared secret if desired, even though it isn't part of the scheme described in the paper.

One of the first proposed R-LWE based signature schemes having provable security while at the same time being practical in the sense that it can be used with similar reliability to that of classical signature schemes like RSA or DSA (and its elliptic curve variants) was the GLP scheme, named after its creators Güneysu, Lyubashevsky, and Pöppelmann, and first introduced in [GLP12].

The scheme proposed is, according to the authors, an improvement from the schemes described in [Lyu09] and [MP11], with an optimization that allows "to reduce signature length by a factor of two" [GLP12]. This optimization is possible via a function that can correctly "compress" the polynomial z_2 from the signature. Figure 2 illustrates the original signature algorithm combining the two schemes from [Lyu09] and [MP11], upon which the optimization improvements take place.

Signing Key: $\mathbf{s}_1, \mathbf{s}_2 \xleftarrow{\$} \mathcal{R}_1^{p^n}$
Verification Key: $\mathbf{a} \xleftarrow{\$} \mathcal{R}^{p^n}, \mathbf{t} \leftarrow \mathbf{a}\mathbf{s}_1 + \mathbf{s}_2$
Cryptographic Hash Function: $\mathrm{H} : \{0,1\}^* \rightarrow D_{32}^n$

$\mathrm{Sign}(\mu, \mathbf{a}, \mathbf{s}_1, \mathbf{s}_2)$
1: $\mathbf{y}_1, \mathbf{y}_2 \xleftarrow{\$} \mathcal{R}_k^{p^n}$
2: $\mathbf{c} \leftarrow \mathrm{H}(\mathbf{a}\mathbf{y}_1 + \mathbf{y}_2, \mu)$
3: $\mathbf{z}_1 \leftarrow \mathbf{s}_1\mathbf{c} + \mathbf{y}_1, \mathbf{z}_2 \leftarrow \mathbf{s}_2\mathbf{c} + \mathbf{y}_2$
4: if $\mathbf{z}_1$ or $\mathbf{z}_2 \notin \mathcal{R}_{k-32}^{p^n}$, then goto step 1
5: output $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{c})$

$\mathrm{Verify}(\mu, \mathbf{z}_1, \mathbf{z}_2, \mathbf{c}, \mathbf{a}, \mathbf{t})$
1: Accept iff
$\mathbf{z}_1, \mathbf{z}_2 \in \mathcal{R}_{k-32}^{p^n}$ and
$\mathbf{c} = \mathrm{H}(\mathbf{a}\mathbf{z}_1 + \mathbf{z}_2 - \mathbf{t}\mathbf{c}, \mu)$

Figure 2: Basic Signature Scheme as described in [GLP12]

The signature produced by the scheme is the triple (z_1, z_2, c). As is described in the paper, where it not for the *hash* function used to verify the signature, sending only polynomials z_1 and c would work in the verification algorithm, because $az_1 + z_2 - tc \approx az_1 - tc$. However, since a *hash* functions must output different values for each different input, no matter how close two different input values are, the verification would fail. The authors' solution involves hashing only the higher order bits of $(ay_1 + y_2)$ and then calculate z_2'. This polynomial represents the changes in the higher order bits of $az_1 + z_2 - tc$ as opposed to $az_1 - tc$, and by sending (z_1, z_2', c) the actual information sent is much smaller, because z_2 needs a lot more bits to be represented. After sending the new signature, the verification process is similar to that of the scheme in figure 2, except that the *hash* function produces the hash based on the higher order bits of the received polynomial instead. Figure 3 shows the algorithm changes in the visual representation of the scheme.

From a efficiency point of view, the authors remark that even though key and signature length may be larger than on some number-theoretic schemes, the process of signing and verifying scales much better. As is the case of most R-LWE schemes, the most costly operation is polynomial multiplication, which as was mentioned before can be accomplished by using FFT or NTT approaches with a $O(n \log n)$ complexity.

Signing Key: $\mathbf{s}_1, \mathbf{s}_2 \xleftarrow{\$} \mathcal{R}_1^{p^n}$
Verification Key: $\mathbf{a} \xleftarrow{\$} \mathcal{R}^{p^n}, \mathbf{t} \leftarrow \mathbf{a}\mathbf{s}_1 + \mathbf{s}_2$
Cryptographic Hash Function: $\mathrm{H} : \{0,1\}^* \rightarrow D_{32}^n$

$\mathrm{Sign}(\mu, \mathbf{a}, \mathbf{s}_1, \mathbf{s}_2)$
1: $\mathbf{y}_1, \mathbf{y}_2 \xleftarrow{\$} \mathcal{R}_k^{p^n}$
2: $\mathbf{c} \leftarrow \mathrm{H}\left((\mathbf{a}\mathbf{y}_1 + \mathbf{y}_2)^{(1)}, \mu\right)$
3: $\mathbf{z}_1 \leftarrow \mathbf{s}_1\mathbf{c} + \mathbf{y}_1, \mathbf{z}_2 \leftarrow \mathbf{s}_2\mathbf{c} + \mathbf{y}_2$
4: if $\mathbf{z}_1$ or $\mathbf{z}_2 \notin \mathcal{R}_{k-32}^{p^n}$, then goto step 1
5: $\mathbf{z}_2' \leftarrow \mathrm{Compress}\ (\mathbf{a}\mathbf{z}_1 - \mathbf{t}\mathbf{c}, \mathbf{z}_2, p, k-32)$
6: if $\mathbf{z}_2' = \perp$, then goto step 1
7: output $(\mathbf{z}_1, \mathbf{z}_2', \mathbf{c})$

$\mathrm{Verify}(\mu, \mathbf{z}_1, \mathbf{z}_2', \mathbf{c}, \mathbf{a}, \mathbf{t})$
1: Accept iff
$\mathbf{z}_1, \mathbf{z}_2' \in \mathcal{R}_{k-32}^{p^n}$ and
$\mathbf{c} = \mathrm{H}\left((\mathbf{a}\mathbf{z}_1 + \mathbf{z}_2' - \mathbf{t}\mathbf{c})^{(1)}, \mu\right)$

Figure 3: Optimized Signature Scheme as described in [GLP12]

In 2013, a new signature scheme was proposed in [Duc+13]. This new scheme introduces a new rejection sampling algorithm that samples from a bimodal Gaussian distribution, effectively allowing for increased performance rivaling other classical schemes. In fact, the authors present a benchmarking experience against the RSA and ECDSA schemes on OpenSSL that compares performance both in the speed of signing and verifying signatures and on the length of keys and signatures produced. The benchmarking is presented on table 1.

Implementation	Security(bits)	Sig. Size (kB)	S. Key size (kB)	P. Key size (kB)	Sign/s	Verify/s
BLISS-0	≤ 60	3.3	1.5	3.3	4k	59k
BLISS-I	128	5.6	2	7	8k	33k
BLISS-II	128	5	2	7	2k	33k
BLISS-III	160	6	3	7	5k	32k
BLISS-IV	192	6.5	3	7	2.5k	31k
RSA-1024	72-80	1	1	1	6k	91k
RSA-2048	103-112	2	2	2	0.8k	27k
RSA-4096	≥ 128	4	4	4	0.1k	7.5k
ECDSA-160	80	0.32	0.16	0.16	17k	5k
ECDSA-256	128	0.5	0.25	0.25	9.5k	2.5k
ECDSA-384	192	0.75	0.37	0.37	5k	1k

Table 1: Table showing the benchmarking of Bliss *versus* RSA and ECDSA for various levels of security (some columns were omitted). The full table is presented in [Duc+13]).

As can be seen, results are relatively on par with the fastest classic algorithms benchmarked, both in key and signature sizes as well as with speed [Akl+16a] [Esp+17]. The apparent downside of the algorithm, which otherwise seems to have the properties to be a strong contender to being standardized, is the sampling over a Gaussian distribution, something already mentioned in this paper. In fact, there exist several weaknesses which can be exploited to recover part of the secret key or, in certain circumstances, even the whole key. These exploits are known as side-channel attacks, which means they do not target the algorithms mathematical basis, targeting instead the actual implementation. Details about the attacks can be consulted in [Esp+17].

In [Akl+16a], the authors propose a new scheme for a lattice-based digital signature, called Ring-TESLA, that offers a stronger security proof. The authors mention that both the GLP and the BLISS schemes offer high performance and provable security. However, the security reductions in each scheme, while valid, are disconnected from the concrete instantiations. When using parameters that are "according to the security reductions" [Akl+16a], the performance gets a significant negative impact, meaning the schemes cannot provide both great performance and a provable security in the instantiation. the Ring-TESLA scheme aims at providing comparable performance to the aforementioned schemes while remaining provably secure in instantiation.

A flaw (independently found by Chris Peikert and Gus Gutoski) in the security reductions present in the Ring-TESLA scheme, as well the follow up Ring-TESLA#, was discovered, which hasn't been addressed since and, while it doesn't seem to be exploitable to create an attack to the scheme, the concrete instantiations lack any formal security proof (as described in [Akl+16b]).

Following the Ring-TESLA scheme, another proposal was made in [Bar+16] in which the authors claim improves on all previous schemes in terms of performance and increased security.

One modification the authors make is similar to that mentioned before when describing the NewHope scheme described in [Alk+15], which consists on generating fresh instances of two polynomials a_1 and a_2 for each key pair generation. By using new polynomials every time a new key pair is created, the risk of the scheme being targeted with an all-for-the-price-of-one attack is greatly reduced, as well as the presence of backdoors that can render the entire scheme insecure.

The key generation scheme in Ring-TESLA has a constraint that rejects the polynomials e_1 and e_2 if at least one of them has a coefficient higher than a parameter L, in which case they have to be sampled again (using a Gaussian distribution). The TESLA# variant completely eliminates such constraints, greatly increasing key generation performance.

In the sign function, one constraint present on the Ring-TESLA scheme was also removed, namely one that restricted the sent of the signature if polynomials w_1 and w_2 have coefficients with absolute value larger than a certain value. Furthermore, as is explained in [Bar+16], the constraint does not offer guaranties that a correct signature is verified as being genuine, and such an event can be observed in practice. The authors provide specific cases where this happens in [Bar+16], and change the constraint so that the rounded values of w_1 and w_2 actually have to be the same as the values of v_1 and v_2, which, according to what the authors say, is the purpose of the initial constraint to begin with. The new constraint also significantly increases the signing success rate, almost doubling that of previous schemes.

It is also worth mentioning that the TESLA# scheme is proven in [Bar+16] to be safe against chosen-message attacks.

The authors also provide two concrete parameter sets, one for providing 128 bit security against classical attacks and another for 128 bits of post-quantum security. Both sets are benchmarked against other algorithms, including GLP and BLISS, as well as classical algorithms. Table 2 shows the results obtained by the authors in [Bar+16].

Scheme	Security (bits)	Sign/s	Verify/s
TESLA#-I	128:64	29.9k	42.1k
TESLA#-II	256:128	12.7k	16.8k
GLP	80:80	7.5k	100.0k
BLISS-0	60:33	4.2k	58.8k
BLISS-I	128:66	8.1k	33.3k
BLISS-II	128:66	2.1k	33.3k
BLISS-III	160:80	4.9k	32.3k
BLISS-IV	192:97	2.7k	31.3k
RSA-1024	80:0	6.0k	90.9k
RSA-2048	112:0	0.85k	26.3k
RSA-4096	144:0	0.11k	7.3k
ECDSA-160	80:0	17.24k	4.9k
ECDSA-256	128:0	9.43k	2.6k
ECDSA-384	192:0	5.13k	1.2k

Table 2: Table showing the benchmarking of Tesla# *versus* RSA, ECDSA, BLISS and GLP. The table is presented in [Bar+16]).

The TESLA# scheme, as can be seen, while providing higher bits of post-quantum security than every other scheme, also manages to be faster in signing operations, showing only a decline in performance in the verification process when using the 128 bit post-quantum security version.

Recently, a new scheme based on the GLP scheme was proposed by Arjun Chopra in [Cho17], named GLYPH. In this paper the author briefly discusses the three schemes described before, namely BLISS, Ring-Tesla (and variants) and GLP, where the different advantages and disadvantages are discussed.

Essentially, the author's remarks are that while the GLP protocol is more efficient than Ring-TESLA, its security is not strong enough it today's standards. The reason the GLP is chosen against BLISS, even though this scheme also has high performance, is because of the Gaussian sampling which is at the heart of the BLISS proposal. The GLP scheme does not sample from a Gaussian distribution.

The GLP scheme got its high performance in part because of the "compressing" of one of the polynomials that would be part of the signature to be sent, which drastically reduced the bit amount necessary to represent said signature. This compression mechanism is changed in this scheme, since the new parameter sets proposed in [Cho17] that are necessary for security nowadays are not compatible with the old compression mechanism, especially the parameter q which is considerably lower than that which was proposed in [GLP12] for the GLP scheme.

Since GLYPH is an "updated" version of the GLP scheme, it makes sense to discriminate the differences introduced in order to improve its security. As was the case with TESLA# in

[Bar+16] and NewHope in [Alk+15], the GLYPH scheme avoids a static public polynomial a, instead giving each certificate root authority the responsibility of choosing and then sharing the public parameters with its users, preventing all-for-the-price-of-one-attacks and backdoors.

The main changes to the scheme come from the new compressing function, which the author calls *compress2*, and a new rounding mechanism (the mechanism that allows the hashing of only the highest bits). The hashing function must, according to the author, output a digest with length of at least 256 bits for 128 bit security, so the one used in the scheme is simply *SHA256*. One interesting detail is that the GLP scheme fixes ω, a parameter that describes how many non-zero coefficients a polynomial c outputted from an encoding function F must have, at 32. GLYPH makes it possible to choose a different value, by using AES in counter mode. By not fixing ω, more combinations of parameters are possible, further increasing the flexibility of this scheme.

The *compress2* function, for a careful choice of parameters, is actually better than the "old" *compress*, being more accurate while at the same time allowing for a wider range of values q in the parameter set.

The scheme's security analysis took into account the optimistic assumptions that the authors of the NewHope scheme took when analyzing their scheme, assuming potential developments in the future that could lead to significant improvements in solving lattice-based problems.

Finally, performance measured by the author against other parameter sets proposed in [GLP12] can be seen in table 3.

	Set I	Set II	GLYPH
m	1024	2048	2048
n	512	1024	1024
q	8383489	16760833	59393
B	16383	32767	16383
ω	32	32	16
Signature size (kB)	0.9	1.9	1.8
Secret key size (kB)	0.1	0.3	0.3
Public key size	1.5	3.1	2.0
Expected number of repetitions	7	7	7
Hamming weight of q	13	11	5
Conservative security level (bits)	< 80	91	137

Table 3: Table showing performance of the GLYPH scheme with two GLP proposed parameter set and then the new set. The table can be seen in [Cho17].

While the GLYPH scheme does show some promise regarding performance and security when compared to previous schemes, it wasn't submitted to NIST, and therefore is not a possible candidate to standardization. Two signature schemes that seem to show more promise, regarding a possible standardization in the future, are qTesla, introduced in [Alk+19] and CRYSTALS-Dilithium, introduced in [Duc+17a]. Both schemes were submitted to NIST and passed the first round. The qTesla scheme is based on the standard R-LWE

problem, while Dilithium is based, like CRYSTALS-Kyber, which was already mentioned in 2.2.1, on the more generic M-LWE problem.

The qTesla algorithm comes as a successor from Ring-Tesla with several modifications the authors explain make the scheme more secure and efficient, such as preventing occasional rejections of valid signatures, providing the scheme's security proof in the Quantum Random Oracle Model (QROM) and, perhaps more importantly, improving its resistance against side-channel attacks, which target certain implementations of Ring-Tesla that do not run in constant time.

Perhaps one of the most important aspects of this scheme is the fact that, while it does use Gaussian sampling to sample polynomials in R, this sampler is constructed in such a way that allows for it to be efficient and at the same time easily implemented to be executed in constant time. Other schemes, such as GLYPH, Dilithium (signatures) or NewHope (key exchange) have a different approach to offer protection against timing attacks, mainly by avoiding Gaussian sampling altogether.

It is noteworthy to point out, however, that the Gaussian sampler is only used to generate the keypair, not being necessary to create or verify signatures. As the authors pointed out, because it is still one of the most, if not the most costly operation in the scheme, avoiding its use in the signing and verification functions, which are the most used functions in a signature scheme, improves significantly the performance of the scheme.

While both schemes were proposed to NIST and passed the same round, and both bear similarities to previous schemes like GLP, there are some core differences that are important to distinguish. First and foremost, Dilithium, as was said before, does not make use of Gaussian sampling. The authors point out that, even though it may be the case that secure instantiations that could be protected against side-channel attacks could be correctly implemented, it is unreasonable to assume that, given its difficulty, every independent implementation of the scheme would be done correctly in order to avoid said vulnerabilities. Therefore, the scheme uses only uniform sampling to generate secret information.

Secondly, Dilithium is based on the more general M-LWE problem, making use of matrices and vectors of polynomials. For example, the key generation process involves generating a $k \times l$ matrix $\mathbf{A}$, for which the authors recommend that $k = 5$ and $l = 4$. The two secret vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ that will be part of the secret key are also vectors of polynomials of, respectively, size l and k. The public key calculated using both secret vectors will also be a vector of polynomials of size k.

Another big difference between the two schemes is the deterministic nature of the Dilithium signing process, while qTesla has a probabilistic signature algorithm [3]. The randomness introduced in a signature with Dilithium comes from part of the private key

3 Even though the deterministic nature of Dilithium was indeed present in the original paper [Duc+17a], a new version [Duc+17b] was submitted after the round 2 results which adds the possibility to randomize the algorithm's signatures. The details will be discussed in 3.

itself, i.e., is different for the same message if the signing keys are different, but is still the same if the key also is the same. The component used from the private key is appended to the message to be signed and serves as input to a collision resistant function (in this case, SHAKE-256) that outputs a value that will be used to produce the signature. Because it is possible that the algorithm must run several times before a valid signature is produced, the authors mention that a counter must also be appended to the input of the SHAKE-256 function so that different outputs come for each iteration. They also mention that the message itself, since it might need several iterations to be signed, is first used to calculate a digest using another collision resistant hash function and then the digest is used instead of the message itself.

Compared to previous schemes with a similar structure, the authors claim that Dilithium manages to distinguish itself by reducing the public key size by a factor of around 2.5, while adding only about 150 bytes to the signature produced as a result. In previous schemes, the public key $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ was calculated and then used posteriorly to verify messages. With Dilithium, $\mathbf{t}$ is split in two components, $\mathbf{t}_1$ and $\mathbf{t}_0$, which are, espectively, the higher and lower bits of $\mathbf{t}$. The public key then only contains $\mathbf{t}_1$, and the secret key contains $\mathbf{t}_0$. By taking away the lower bits from $\mathbf{t}$, then signature must contain additional information that compensates the absence of those bits. That information is calculated in the signing process and is called a ”hint”, and, if included in the signature, makes up for the absence of the lower bits of $\mathbf{t}$.

In [Alk+19], the authors provide a table comparing different algorithms in regards of their key and signature sizes and cycle counts for performing their signing and verification algorithms. With recommended parameters, to achieve a post-quantum security of around 160 bits, Dilithium manages to achieve similar results to qTesla for certain parameter sets.

2.3 Related Work

Post-quantum cryptography is of great importance for the future of cryptography, and secure and efficient quantum-safe algorithms capable of ”replacing” classical number-theoretic schemes are needed to ensure the transition to the post-quantum era can be accomplished without compromising the future of communications.

Careful consideration of which algorithms actually fit this description, and then proceeding to a good implementation of said algorithm is probably the best way to find out how exactly the post-quantum cryptographic schemes fare against the more established cryptographic algorithms. In this sense, several authors have already tried to implement algorithms of their own, or algorithms described by other authors, and insert them into the TLS protocol, usually via OpenSSL.

The NewHope scheme [Alk+15] was actually implemented by Google on the TLS protocol [Bra16; Lan16] in an experiment to check the viability of a new post-quantum scheme in the real-world context, as well as to further gather attention to the importance of the subject. The experiment was conducted on a small percentage of devices using Chrome Canary and on top of the X25519 elliptic curve scheme (this hybrid variant was called "CECPQ1") to safeguard against potential vulnerabilities with the NewHope scheme. The experiment ended in November 2016 [Lan16], and NewHope wasn't found to have any negative impact aside from a slightly bigger latency in communications, which were considered to be almost exclusively caused by bigger message sizes.

Bos et al. [Bos+14] implemented the scheme introduced in [DXL12] and mentioned in 2.2.1 on the TLS protocol via integration with the OpenSSL library. It is in this paper that the importance of a scheme that shares a structure with the classical DH-like schemes became evident, since it was mentioned by the authors that such a scheme is advantageous for integration purposes. The integration comes with the main objective of testing a post-quantum algorithm against other well established classical algorithms, in this case the ECDH (Elliptic Curve Diffie-Hellman) schemes, so that it can be shown in real-world usage that right now it is already possible to use post-quantum schemes in practice.

Butin, Wälde, and Buchmann [BWB18] implemented, in a similar endeavor, a hash-based post-quantum signature scheme on TLS via OpenSSL to build trust on post-quantum schemes by showing their performance in real-life usage. As a accompanying objective, the authors also provide a basic understanding of how the OpenSSL library works. However, the information provided is scarce when it comes to actual instructions on how to implement new algorithms.

Specifically relating to the integration part now, the work done in [Jur12] sees the full integration of a symmetric cipher called Indect Block Cipher, with a detailed analysis of how to accomplish the integration part. There are, however, two aspects to take into account. Firstly, while bearing similarities, there are some differences between integrating symmetric and asymmetric ciphers, including the OpenSSL's high level interface EVP that changes for different types of cryptographic functionalities. Secondly, the implementation was accomplished in 2012. Since then, several changes were made to the OpenSSL source code, possibly changing the way certain components interconnect and thus requiring a deep code analysis to understand what changed. A very succinct and compact version of this paper by the same author is presented in [JN16].

Still focusing only on the TLS integration, Käsper [Käs12] introduces an optimized implementation of the NIST P-224 elliptic curve on OpenSSL. The aspects to take into account here, especially regarding the OpenSSL version at the time of inclusion, are similar to those mentioned above about the work in [Jur12].

Finally, in [SM16], as was already briefly mentioned, Stebila and Mosca introduce a new framework inspired on a previous intermidiate API developed by them to simplify the process of integrating new algorithms on OpenSSL. The project, called Open Quantum Safe Project, aims at facilitating the test of post-quantum algorithms in an experimental fashion, and includes a fork with the OpenSSL project where the authors have included all necessary modification to allow for an easier integration of new algorithms, provided said algorithms were added to the Open Quantum Safe library liboqs first.

3

PRELIMINARIES

This chapter aims at presenting some mathematical notations relevant to better understand the schemes to be implemented. In chapter 2, the LWE, and consequently, the R-LWE problems were repeatedly mentioned, yet no definitions were provided that explain what exactly these problems are, as well as definitions of some important mathematical notions that the problems involve. Furthermore, and even though the previous chapter briefly introduced the NewHope key exchange, the GLYPH digital signature and the Dilithium digital signature, the algorithms themselves were not properly described, nor were its main components, such as functions that are necessary for their implementation. This chapter aims at defining said algorithms as they are defined by the authors that created them.

3.1 NOTATION

This section aims at fixing some mathematical notations used throughout the paper.

For any prime number q, $\mathbb{Z}_q$ denotes the finite field of integers modulo q. The dot product between two vectors is represented by $\langle a, b \rangle$.

Every uni-variate polynomial presented is of the form

$$a_0X^0 + a_1X^1 + a_2X^2 + ... + a_{n-1}X^{n-1}$$

for a certain value n.

Each element of the ring $R_q \equiv \mathbb{Z}_q[X]/(X^n + 1)$ is uniquely represented by a polynomial with coefficients $a_i \in \mathbb{Z}_q$ and degree less than n.

Whenever some variable a is sampled using the notation $a \xleftarrow{\$} \mathbb{Z}_y^x$, it means it is uniformly sampled from $\mathbb{Z}_y^x$ (following the notation present in most papers).

3.2 THE LEARNING WITH ERRORS PROBLEM

The Learning With Errors problem is the problem of finding a secret vector s uniformly sampled from $\mathbb{Z}_q^n$, for integer values n and q, given multiple pairs $(a, b) = (a, \langle a, s \rangle + e)$,

where a is a uniformly sampled vector in $\mathbb{Z}_q^n$ and e a noise value in $\mathbb{Z}$ sampled according to a Gaussian distribution $N(\mu, \sigma^2)$, with $\sigma = \alpha * q$ and $\mu = 0$.

The problem can be seen as an adversary requesting an oracle (the LWE oracle) for x different and independent samples (a, b) and then being capable of determining the secret s from those samples. This problem has been shown to be at least as hard to solve as worst-case lattice problems. As is stated in [Reg05], each sample represents a random linear equation in the form

$$a_0 s_0 + a_1 s_1 + a_2 s_2 + ... + a_{n-1} s_{n-1} \approx y \mod q \tag{2}$$

, where a_i and s_i are the components of the vectors a and s, respectively (this equation represents the dot product of these two vectors). The value y is an approximation because the result of the dot product is perturbed by the small value e sampled from the Gaussian distribution $N(\mu, \sigma^2)$. Regev further explains that the noise value e is crucial for the hardness of the problem, since without it the problem could easily be solved by using Gaussian elimination given a sufficient number of different linear equations. By attempting to solve the equations using this technique, the small noise values in each sample are greatly amplified, rendering the resulting information basically useless. Reducing noise amplification when applying row elimination is the focus of an attack algorithm to the LWE problem, called BKW, which essentially tries to apply multiple eliminations with a single row addition [Alb+12], thus avoiding amplifying noise with more additions than necessary. Another, more efficient variant, inspired by the former, can be consulted in [DTV15].

3.3 The Ring - Learning With Errors Problem

The R-LWE is a variant from the LWE problem where instead of sampling vectors in a finite field $\mathbb{Z}_q^n$ for positive integers n, q, the sampling is usually done from a polynomial ring $R_q = \mathbb{Z}_q[X]/(X^d + 1)$, where usually $q \equiv 1 \mod 2d$ and d is of the form $d = 2^k$, for $k > 0$ (although d can also be a prime, polynomial multiplication using the number-theoretic transform is significantly more efficient if d is a power of 2).

Similarly to the LWE problem, there is a Gaussian distribution $N(\mu, \sigma^2)$ with $\mu = 0$ on $\{\frac{-(q-1)}{2}, ..., \frac{(q-1)}{2}\}$ used to sample polynomials with "small" coefficients.

An adversary then asks the oracle for samples of the form $(a(x), b(x)) = (a(x), a(x) * s(x) + e(x))$, where $a(x)$ is a polynomial uniformly sampled from R_q (each coefficient has the same probability of being sampled) and $s(x), e(x)$ have "small" coefficients sampled from the Gaussian distribution. The multiplication between $a(x)$ and $s(x)$ is the standard polynomial multiplication. The problem is then to be able to recover $s(x)$ given the samples requested from the oracle. The problem has been proven to be as hard as worst-case problems in ideal lattices in [LPR10].

3.4 THE MODULE - LEARNING WITH ERRORS PROBLEM

The M-LWE problem, which was introduced by Brakerski, Gentry, and Vaikuntanathan in [BGV11] (and was initially called General - Learning With Errors Problem) is, in essence, a generalized version for both R-LWE and LWE.

As is described in [BGV11], it is the problem of finding a secret vector s uniformly sampled from R_q^n given multiple tuples $(a, b) = (a, \langle a, s \rangle + e)$, where a is uniformly sampled from R_q^n, e is sampled from a distribution χ over $\mathbb{Z}[X]/(X^d + 1)$ and $R_q = \mathbb{Z}_q[X]/(X^d + 1)$, for a given dimension n, prime q and degree d.

As is pointed out in [BGV11], the LWE problem is the M-LWE problem with $d = 1$, i.e., where $R_q = \mathbb{Z}_q[X]/(X + 1)$. Similarly, the R-LWE problem is a specific case of M-LWE where $n = 1$, and s and a are sampled from R_q^1.

In [AD17], Albrecht and Deo regard the M-LWE problem as a sort of compromise between the efficiency of R-LWE and the added confidence in the security of the LWE problem. In fact, as was mentioned before in 2.1.2, and as is said in [AD17; Bos+16], general lattices and, subsequently, hard problems on general lattices have been more thoroughly studied than those in ideal lattices, and it is not known if the extra algebraic structure present in the R-LWE problem makes the problem easier to solve. It could be the case, according to the authors, that the M-LWE problem can offer a better level security than R-LWE and better performance than LWE. Furthermore, security in schemes based on this problem, like Dilithium, can vary by changing the dimension n used in the scheme, as opposed to varying the polynomial degree (because in most cases the degree is a power of two, there are limited degree values that can be used to tweak the scheme's security). In fact, Dilithium offers different dimensions for different security levels, while maintaining the polynomial degree contant for all parameter sets.

3.5 GENERAL R-LWE KEY EXCHANGE FRAMEWORK

Most key exchange schemes proposed based on the LWE problem have three main components in common: a ring instantiation, an error probability distribution and a way to encode and decode information to achieve the same shared secret for both parties (like the reconciliation approaches already mentioned in chapter 2). Most schemes based on R-LWE and M-LWE use a polynomial ring to sample public parameters, errors and the secret key, although it isn't strictly necessary to use such a ring; in fact, some authors, like Mike Hamburg [Ham19] and Gu Chunsheng [Chu17] have proposed schemes that are based on different rings, and Chunsheng proved that R-LWE schemes based on integer rings, as opposed to polynomials ones, can offer similar security.

In [Ham19], the author has proposed a key exchange scheme called ThreeBears where the underlying problem is instantiated with a ring of integers modulo a Mersenne number $N = P(q)$, for q and $P(x)$ a polynomial of degree D. The scheme, which is based on NewHope and Kyber, was submitted to NIST and passed to the second round.

In general, to instantiate a R-LWE key exchange scheme, it's necessary to instantiate the three components described above. Specifically, it's necessary to instantiate a ring R with dimension d and an error distribution on R. Most schemes described either use a Gaussian [Bos+16; Bos+14] or binomial [Alk+15; Bos+17] distribution over R. Finally, the scheme must also propose a concrete way to encode and decode the information to create a shared key. Figure 4, presented in [Ham19], illustrates the general M-LWE key exchange scheme.

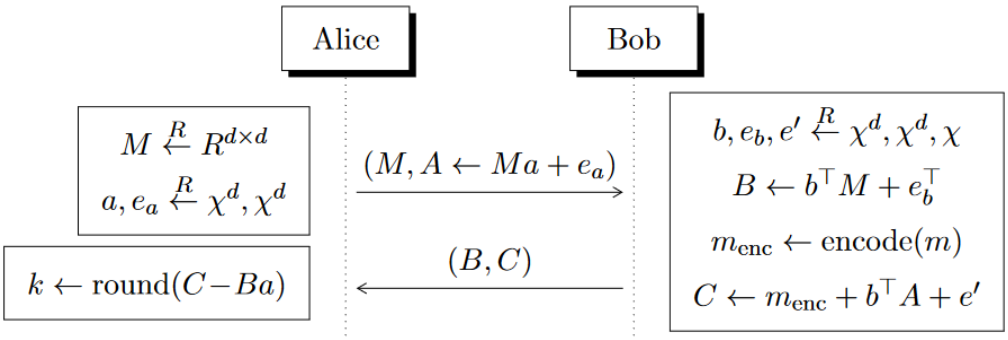


Figure 4: Image showing a general framework for M-LWE key exchange. The image was taken from [Ham19].

The scheme presented in the image is generalized for a value d, ring R, distribution χ and functions *encode* and *round*. When using a polynomial ring $R_q = \mathbb{Z}_q[X]/(X^D + 1)$ with dimension $d = 1$ and degree D, matrix M becomes a 1×1 matrix, i.e., M is essentially a polynomial in R_q, and so are values a and e_a, sampled from ring R_q with a certain probability distribution. Therefore, when $d = 1$ and R is the described ring of polynomials, $A = Ma + e_a$ is a polynomial, which in the proposed schemes usually represents Alice's public key. Bob's B follows the same logic.

Each algorithm can describe its own strategy to derive an exact shared secret k from the approximate values both Alice and Bob obtain using each other's public keys. As an example, in the NewHope scheme variant proposed in [Alk+16], Bob encodes the random value m represented in the image by encoding each of its bits into coefficients of a polynomial with 1024 coefficients (mapping 1 bit into four coefficients), and the round function Alice uses is the decoding function that recovers the secret m from the ciphertext by recovering each bit previously encoded.

3.6 THE NEWHOPE KEY EXCHANGE

Of all key exchange schemes mentioned in chapter 2, the NewHope scheme (and its simple variant) is arguably one of the best suited for integration. It checks most conditions for a good algorithm to effectively "replace" a classical algorithm on TLS, except, in some extent, for the simplicity of the algorithm. In fact, the authors themselves created a simplified version with no cost on security exactly to provide a variant that is simpler to implement, because it no longer uses the reconciliation mechanism of the NewHope scheme.

There is also the fact that it was already integrated by Google on its post-quantum experiment, and that its precursor, the BCNS scheme from [Bos+14], was already successfully integrated on TLS by its authors to test its viability. Furthermore, it is quite simple to describe when compared with other algorithms like Kyber.

Alkim et al. propose a concrete set of parameters for instantiation, making use of a polynomial ring R_q as defined in section 3.3, where $q = 12289$ and $n = 1024$.

The error distribution from where "small" polynomial are sampled, as was pointed out before, is not Gaussian, but instead a centered binomial distribution with parameter $k = 16$ and variance $\sigma^2 = \frac{k}{2}$. The authors prove in [Alk+15] that for the same standard deviation ($\sqrt{16/2}$), the effect on security from using a binomial distribution instead of a rounded Gaussian is negligible.

Table 4 shows the whole key exchange scheme as described by the authors in [Alk+15].

Parameters: $q = 12289 < 2^{14}, n = 1024$ Error distribution: ψ_{16}		
Alice		**Bob**
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow Parse(SHAKE128(seed))$		
$\mathbf{s,e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow Parse(SHAKE128(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} HelpRec(\mathbf{v})$
$v \leftarrow Rec(\mathbf{v}', \mathbf{r})$		$v \leftarrow Rec(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow SHA3\text{-}256(v)$		$\mu \leftarrow SHA3\text{-}256(v)$

Table 4: The proposed NewHope key exchange scheme in [Alk+15].

Every variable in bold notation denotes a polynomial, so as to follow the notation used in [Alk+16].

The polynomial **a**, which is generated for each session, results from the parsing of the output of the extendable output function (XOF) SHAKE128 mentioned before. The function

Parameters: $q = 12289 < 2^{14}, n = 1024$ Error distribution: ψ_{16}		
Alice		**Bob**
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\hat{\mathbf{a}} \leftarrow Parse(SHAKE128(seed))$		
$\mathbf{s},\mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s},\mathbf{e}',\mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\hat{\mathbf{s}} \leftarrow NTT(\mathbf{s})$		
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + NTT(\mathbf{e})$	$\xrightarrow{m_a = encodeA(seed, \hat{\mathbf{b}})}$	$(\hat{\mathbf{b}}, seed) \leftarrow decodeA(m_a)$ $\hat{\mathbf{a}} \leftarrow Parse(SHAKE128(seed))$ $\hat{\mathbf{t}} \leftarrow NTT(\mathbf{s}')$ $\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + NTT(\mathbf{e}')$ $\mathbf{v} \leftarrow NTT^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$
$(\hat{\mathbf{u}}, \mathbf{r}) \leftarrow decodeB(m_b)$	$\xleftarrow{m_b = encodeB(\hat{\mathbf{u}}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} HelpRec(\mathbf{v})$
$\mathbf{v}' \leftarrow NTT^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		
$v \leftarrow Rec(\mathbf{v}', \mathbf{r})$		$v \leftarrow Rec(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow SHA3\text{-}256(v)$		$\mu \leftarrow SHA3\text{-}256(v)$

Table 5: A more complete description of the NewHope scheme with NTT and encoding functions (also presented in [Alk+15])

.

outputs 16-bit integers which are then either accepted as a valid number for a coefficient of $\mathbf{a}$, if the value is smaller then $5q$, or rejected otherwise.

The public key $\mathbf{b} = \mathbf{a} * \mathbf{s} + \mathbf{e}$, where $\mathbf{s}, \mathbf{e}$ were sampled according to the binomial distribution, is sent to the other peer (Bob), as well as the seed used to generate polynomial $\mathbf{a}$.

The other peer (Bob) then calculates $\mathbf{u} = \mathbf{a}\mathbf{s}' + \mathbf{e}'$, its public key, and $\mathbf{v} = \mathbf{b}\mathbf{s}' + \mathbf{e}''$, its approximate shared key, where $\mathbf{s}', \mathbf{e}', \mathbf{e}'$ are also sampled in the binomial distribution. The exact shared key is calculated by using the reconciliation mechanism aforementioned, first by calculating the reconciliation information $\mathbf{r}$ and sending both $\mathbf{u}$ and $\mathbf{r}$ back to Alice. Alice then computes $\mathbf{v}' = \mathbf{u}\mathbf{s}$ and then retrieves v by using $\mathbf{r}$ and $\mathbf{v}'$ as parameters for the reconciliation function. Bob uses his approximate key $\mathbf{v}$ and the same value $\mathbf{r}$ to derive the same exact value v. The shared key is finally calculated by hashing v with SHA-256.

A more complete and implementation oriented description of the protocol is presented in table 5. Whenever possible, polynomials are kept in their NTT format, so as to increase the overall efficiency of the scheme. The polynomial a can also be sampled directly in the NTT format, increasing performance by skipping the need to compute the transform after sampling.

The NewHope-Simple scheme is identical in most ways, but has a different mechanism for deriving an exact key, greatly simplified in this variant. Instead of using functions *Rec* and *HelpRec* to achieve a shared key, it relies on 4 functions: *NHSEncode*, *NHSDecode*, *NHSCompress* and *NHSDecompress*. Definitions of these functions as presented in [Alk+16] can be seen in algorithms 1, 2, 3 and 4, respectively.

Algorithm 1 NHSEncode

Input: $v \in \{0,1\}^{256}$
Output: $k \in R_q$

1: $k \leftarrow 0$
2: **for** $i = 0, i < 256$ **do**
3: $\quad k_i \leftarrow v_i.\lfloor \frac{q}{2} \rfloor$
4: $\quad k_{i+256} \leftarrow v_i.\lfloor \frac{q}{2} \rfloor$
5: $\quad k_{i+512} \leftarrow v_i.\lfloor \frac{q}{2} \rfloor$
6: $\quad k_{i+768} \leftarrow v_i.\lfloor \frac{q}{2} \rfloor$
7: **end for**
8: **return** k

Algorithm 2 NHSDecode

Input: $k' \in R_q$
Output: $v \in \{0,1\}^{256}$

1: $v \leftarrow 0$
2: **for** $i = 0, i < 256$ **do**
3: $\quad t \leftarrow \sum_{j=0}^{3} |v_{i+256.j} - \lfloor \frac{q}{2} \rfloor|$
4: $\quad$ **if** $t < q$ **then**
5: $\quad\quad v_i \leftarrow 1$
6: $\quad$ **else**
7: $\quad\quad v_i \leftarrow 0$
8: $\quad$ **end if**
9: **end for**
10: **return** v

Algorithm 3 NHSCompress

Input: $c \in R_q$
Output: $\bar{c} \in \{0,...,7\}^{1024}$

1: **for** $i = 0, i < 1024$ **do**
2: $\quad \bar{c}_i \leftarrow \lfloor (c'_i.8)/q \rceil \mod 8$
3: **end for**
4: **return** $\bar{c}$

Algorithm 4 NHSDecompress

Input: $\bar{c} \in \{0,...,7\}^{1024}$
Output: $\bar{c} \in R_q$

1: **for** $i = 0, i < 1024$ **do**
2: $\quad c'_i \leftarrow \lfloor (\bar{c}_i.q)/8 \rceil$
3: **end for**
4: **return** c'

While the essential information about the purpose of these functions was mostly given in chapter 2, it is provided again here for convenience.

The *NHSEncode* function encodes a single bit into 4 polynomial coefficients, effectively "transforming" a 256-bit value in a polynomial in R_q. The opposite function *NHSDecode* takes the same four coefficients and maps them into a single bit again.

The *NHSCompress* function takes as input a polynomial in R_q and outputs another polynomial with coefficients modulo 8, and its counterpart *NHSDecompress* does the exact opposite, outputting a polynomial in R_q using a polynomial with coefficients modulo 8 as input.

3.7 THE GLYPH SIGNATURE SCHEME

The GLYPH scheme proposed in [Cho17] is an improvement on the older GLP scheme from [GLP12]. The improvements come from a new set of parameters that provide 128 bits of security, and a new compressing and rounding function.

Parameter choice for the GLYPH scheme is as follows: $n = 1024, q = 59393, B = 16383$ and $\omega = 16$.

As with most signature schemes, GLP/GLYPH is composed of three main functions: Key Generation, Sign and Verify. The pseudo-code, as described in [Cho17], is presented respectively in algorithms 5, 6 and 7.

Algorithm 5 Key Generation

Input: $a \in R_q$
Output: Key pair (r, t) (r is private and t is public)

1: $s, e \xleftarrow{\$} R_{q,1}$
2: $t \leftarrow a * s + e$
3: $r \leftarrow (s, e)$
4: **return** (r, t)

The polynomials $\mathbf{s}, \mathbf{e}$ are uniformly sampled from $R_{q,1}$, which means the absolute value of each coefficient must not exceed, in this case, 1, when considering the coefficients in the form of $[-\frac{(q-1)}{2}, \frac{(q-1)}{2}] \mod q$.

All other operations, such as polynomial multiplication, are the same as with the NewHope scheme.

Functions F and H are essential to ensure the scheme behaves correctly. H represents a hash function which must output a string of a fixed length for any arbitrary output, while being capable of securing at least 128 bits of security. The authors use SHA-256 for this purpose, since it matches these requirements. F is an encoding function that maps the result from the hash function to a polynomial which has at most ω coefficients different than zero,

Algorithm 6 Sign

Input: Message μ, private key (s, e), public a, public key t
Output: Signature (z_1, z_2, c)

1: $y_1, y_2 \xleftarrow{\$} R_{q,B}$
2: $c' \leftarrow H(\lfloor a * y_1 + y_2 \rfloor_{B-\omega} | \mu)$
3: $c \leftarrow F(c')$
4: $z_1 \leftarrow s * c + y_1$
5: $z_2 \leftarrow e * c + y_2$
6: **if** $(\|z_1\|_\infty \, or \, \|z_2\|_\infty) > (B - \omega)$ **then**
7: Start over from step 1
8: **end if**
9: $z_2 \leftarrow Compress2((a * z_1 - t * c), z_2)$
10: **return** (z_1, z_2, c)

with values either -1 or 1. As was mentioned in chapter 2, in the original scheme [GLP12] the authors used a fixed value $\omega = 32$ and outputted the resulting polynomial using a specific function F for the purpose. The authors in [Cho17] used a different approach (using AES in counter mode) that allowed them to use a variable ω, making the scheme more flexible.

Algorithm 7 Verify

Input: Message μ, signature (z_1, z_2, c), public a, public key t
Output: $v \in \{0, 1\}$

1: **if** $(\|z_1\|_\infty \, or \, \|z_2\|_\infty) > (B - \omega)$ **then**
2: **return** 0
3: **end if**
4: $d' \leftarrow H(\lfloor a * z_1 + z_2 - t * c \rfloor_{B-\omega} | \mu)$
5: $d \leftarrow F(d')$
6: **if** $(d = c)$ **then**
7: **return** 1
8: **else**
9: **return** 0
10: **end if**

The pseudo-code from the compressing function, called *compress2* by the authors so as to distinguish it from the *compress* function from the GLP scheme, as well as for auxiliary function *CompressCoefficient*, can be seen respctively in algorithms 8 and 9. Both algorithms are present in [Cho17].

The function $\lfloor . \rfloor_K$ (*K-floor function*), as is defined by the authors, is to be used instead of any rounding in K present in the GLP scheme. For any integer x, $\lfloor x \rfloor_K = (x \mod q)/(2K + 1)$. Using the *K-floor function* on a polynomial is equivalent to using it on each coefficient individually.

Algorithm 8 Compress2

Input: $y \in R_q, z \in R_{q,K}$
Output: $z' \in R_{q,K}$
1: **for** $i = 0, i < n$ **do**
2: $\quad z'_i = CompressCoefficient(y_i, z_i)$
3: **end for**
4: **return** z'

Algorithm 9 CompressCoefficient

Input: $y \in [0, q[, z \in [-K, K]$
Output: $z' \in \{-K, 0, K\}$
1: **if** $(\lfloor y + z \rceil_K) = \lfloor y \rceil_k$ **then**
2: $\quad$ **return** 0
3: **end if**
4: **if** $(y \in [0, K[)$ **then**
5: $\quad$ **return** -K
6: **end if**
7: **if** $(y \in [q - K, q[)$ and $(z > 0)$ **then**
8: $\quad$ **return** K
9: **end if**
10: **if** $(\lfloor y + z \rceil_K) < \lfloor y \rceil_k$ **then**
11: $\quad$ **return** -K
12: **end if**
13: **return** K

3.8 THE DILITHIUM SIGNATURE SCHEME

When it comes to integrating signature algorithms that were submitted to NIST, and therefore are viable candidates for future standardization, there are two schemes based on the LWE problem that currently pose as good options: qTesla and Dilithium. The reason Dilithium was chosen instead of qTesla as the algorithm to integrate on OpenSSL is simply because qTesla was already implemented in the master branch of the Open Quantum Safe Project. By adding Dilithium, these two LWE based schemes currently on the second round of the NIST post-quantum project can be used and tested on TLS using the Open Quantum Safe Project.

Dilithium is defined by a set of parameters that vary depending on the security level. The recommended set, as is said in the Dilithium web page, provides around 128 bits of security for all known classical and quantum attacks. The authors point out in [Duc+17a] that some elements, such as the ring from which every polynomial is sampled, are always the same for each level of security. The parameter $q = 2^{23} - 2^{13} + 1 = 8380417$, as well as parameters that directly depend only on q, are also constant for every security level.

The recommended parameter set to be used with Dilithium, in order to achieve about 128 bits of security, is represented in table 6:

Parameter	Value	Description
n	256	Degree of polynomial ring
q	8380417	Ring modulus
d	14	Parameter used to divide public key t into $\mathbf{t}_1$ and $\mathbf{t}_0$
weight of c	60	Number of non-zero coefficients (± 1) in polynomial c
$\gamma_1 = (q-1)/16$	523776	Upper-bound for coefficients of polynomials in $\mathbf{v}$
$\gamma_2 = \gamma_1/2$	261888	Upper-bound used when calculating $\mathbf{w}_1$
(k,l)	(5,4)	Dimensions for public matrix $\mathbf{A}$
η	5	Upper-bound for the every polynomial coefficient in $\mathbf{s}_1$ and $\mathbf{s}_2$
β	275	Upper-bound for the coefficient of $c\mathbf{s}_i$, with $i \in 1, 2$
ω	96	Upper-bound for the number of 1's in the hint
Public key	1472	Public key size in bytes
Secret key	3504	Secret key size in bytes
Signature	2701	Signature size in bytes

Table 6: Table showing the values for Dilithium's recommended parameter set, as well as key and signature sizes. A more complete table, showing parameters for each security level, as well as information regarding performance and bit security, can be consulted in [Duc+17a].

The main functions in Dilithium, namely Key Generation, Sign and Verify, can be seen in algorithms 10, 11 and 12 respectively.

Key generation involves, essentially, generating a random matrix $\mathbf{A}$ with dimensions $k \times l$, two vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ and using those elements to produce public key $\mathbf{t}$. As was mentioned previously in 2.2.2, each of these elements has polynomials in R_q as coefficients, instead of elements of $\mathbb{Z}_q$.

Matrix $\mathbf{A}$ is generated as the output of the *ExpandA* function, which takes a 256-bit seed ρ and generates polynomials in their NTT domain.

Algorithm 10 Key Generation

Input:
Output: Keypair (pk, sk)

1: $\rho \xleftarrow{\$} \{0,1\}^{256}$
2: $K \xleftarrow{\$} \{0,1\}^{256}$
3: $(\mathbf{s}_1, \mathbf{s}_2) \xleftarrow{\$} S_\eta^l \times S_\eta^k$
4: $\mathbf{A} \in R_q^{k \times l} := ExpandA(\rho)$
5: $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) := Power2Round_q(\mathbf{t}, d)$
7: $tr \in \{0,1\}^{384} := CRH(\rho || \mathbf{t}_1)$
8: **return** $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

The *Power2Round* function is responsible for dividing vector $\mathbf{t}$ into its high order and low order bits, producing, respectively, $\mathbf{t}_1$ and $\mathbf{t}_0$. Using $\mathbf{t}_1$ and ρ, a new pseudo-random 384-bit value tr is produced, as the output of a collision-resistant hash function CRH. That value partially responsible for the randomness of the signature scheme, for a certain pair of keys. The is also another pseudo-random seed K generated in the key generation process, which will be used as another source of randomness for the signing process.

The signature algorithm (11) receives the message to be signed and the secret key and outputs a signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$.

As is the case for every function, matrix $\mathbf{A}$ is generated from the seed ρ, which comes as a part of the secret key (but also the public key). The message is then concatenated to tr that is part of the secret key, and the result serves as input for the collision-resistant hash function CRH, which outputs a 384-bit value μ. This value is dependent on the message itself, and is always the same for the same secret key and message.

After obtaining μ, the next step is generating vector $\mathbf{v}$, from which every coefficient is an element of $S_{\gamma_1 - 1}$. The function *ExpandMask* receives as input the 256-bit pseudo-random value K, which is part of the secret key, concatenated with μ and a counter, incremented with each unsuccessful signature, to produce the vector $\mathbf{v}$. Internally, it uses SHAKE-256 to generate coefficients for each polynomial in $\mathbf{v}$. Because the counter value is never takes the same value twice, signatures produced by different iterations will also be different, until one is considered valid. In the updated version of the paper [Duc+17b], the possibility to produce randomized signatures was introduced. In this variant, the only modification to the algorithm is in the signing procedure, which, instead of having the concatenation of K and μ as input to *ExpandMask*, the function receives either the result of first passing the concatenation through a collision-resistant hash function (non-randomized version, since it still depends on the key and message) or a completely random string of bits 384 bits (randomized version). The altered parts can be seen in algorithm 13.

Algorithm 11 Sign

Input: Secret key sk, message M
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$

$\mathbf{A} \in R_q^{k \times l} := ExpandA(\rho)$
$\mu \in \{0,1\}^{384} := CRH(tr||M)$
$\kappa := 0, (\mathbf{z}, h) := \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} \in S^l_{\gamma_1 - 1} := ExpandMask(K||\mu||\kappa)$
  $\mathbf{w} := \mathbf{A}\mathbf{y}$
  $\mathbf{w}_1 := HighBits_q(\mathbf{w}, 2\gamma_2)$
  $c \in B_{60} := H(\mu||\mathbf{w}_1)$
  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
  $(\mathbf{r}_1, \mathbf{r}_0) := Decompose_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ **then**
    $(\mathbf{z}, \mathbf{h}) := \perp$
  else
    $\mathbf{h} := MakeHint_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$
    if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or # of 1's in $\mathbf{h} > \omega$ **then**
      $(\mathbf{z}, \mathbf{h}) := \perp$
    end if
  end if
  $\kappa = \kappa + 1$
end while
return $\sigma = (\mathbf{z}, \mathbf{h}, c)$

The *Decompose* function is another function used to generate the higher and lower order bits of a given element in $\mathbb{Z}_q$, and it's used in, for example, the *HighBits* and *LowBits* functions. It is also used to generate the high and low order bits of $\mathbf{w} - c\mathbf{s}_2$. The high order bits must be equal to the high order bits of $\mathbf{w}$, calculated by the *HighBits* function.

The *MakeHint* and, as a counterpart, *UseHint*, respectively generate the hint vector $\mathbf{h}$ from $\mathbf{w} - c\mathbf{s}_2$ and use the hint to recover the higher bits of $\mathbf{Az} - c\mathbf{t}_1$. As is pointed out by the authors, $\mathbf{w} - c\mathbf{s}_2 = \mathbf{Az} - c\mathbf{t}$, and so the higher bits of $\mathbf{w}$ must be the same as the higher bits of $\mathbf{Az} - c\mathbf{t}$. Since in this scheme $\mathbf{t}$ is no longer present, the hint allows to recover the missing lower bits $\mathbf{t}_0$ and calculate the high bits of $\mathbf{Az} - c\mathbf{t}_1$.

Algorithm 12 Verify

Input: Public key pk, message M, signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
Output: Accept or Reject
1: $\mathbf{A} \in R_q^{k \times l} := ExpandA(\rho)$
2: $\mu \in \{0,1\}^{384} := CRH(CRH(\rho||\mathbf{t}_1)||M)$
3: $\mathbf{w}_1' := UseHint_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
4: **return** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c = H(\mu||\mathbf{w}_1')$ and # of 1's in $\mathbf{h} \leq \omega$

Algorithm 13 Sign (changes from new version)

Input: Secret key sk, message M
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
1: ...
2: $\kappa := 0, (\mathbf{z}, h) := \perp$
3: **if** *randomize* **then**
4: $\quad \rho' \xleftarrow{\$} \{0,1\}^{384}$
5: **else**
6: $\quad \rho' \in \{0,1\}^{384} := CRH(K||\mu)$
7: **end if**
8: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
9: $\quad \mathbf{y} \in S^l_{\gamma_1 - 1} := ExpandMask(\rho'||\kappa)$
10: $\quad$ ...
11: **end while**
12: ...

The verification function (12) simply generates matrix $\mathbf{A}$ and μ from the public key (which contains ρ and $\mathbf{t}_1$) and the message, and then tries to calculate $\mathbf{w}_1'$ as the higher bits of $\mathbf{Az} - c\mathbf{t}_1$, using the hint received in the signature as was mentioned above. The signature is accepted if no coefficient in $\mathbf{z}$ is larger than the specified bound $\gamma_1 - \beta$, the number of 1's in $\mathbf{h}$ doesn't exceed a certain bound as well and if the has used to calculate c in the signature function outputs the same c when given $\mathbf{w}_1'$ concatenated with μ as input.

4

INTEGRATION ON THE OPEN QUANTUM SAFE PROJECT

The Open Quantum Safe Project [SM16] is an open source project that aims at testing different post-quantum key exchange and signature algorithms, by providing a library that can be used in general applications and integrated in the TLS protocol to test its usability.

The project is divided in essentially two distinct parts: the general purpose cryptographic library liboqs, which contains a consistent API for every implemented post-quantum algorithm, and can be used as a regular C library in most applications, and the integration of this library in other projects, such as OpenSSL and OpenSSH. In order to integrate a new post-quantum algorithm on OpenSSL using the Open Quantum Safe liboqs library, it's necessary to first add the new algorithm to liboqs and then modify the OpenSSL source code with the necessary changes. It is noteworthy that the OpenSSL source code used for the implementation is not the original, unmodified source code from the OpenSSL project's master fork, instead being a different fork with several modifications from the Open Quantum Safe Project authors. However, these modifications do not alter any of the already established OpenSSL core functionalities, adding instead new ones from the liboqs library.

Open Quantum Safe already integrates several post-quantum algorithms in its current version, both key exchange schemes and digital signatures (depending on the fork). As a requisite to add a new algorithm, it must have been submitted to the NIST Post-Quantum Cryptography Standardization Process, as well as have passed to the current round, which means that every algorithm already integrated in the current version meets these criteria.

Currently, two different branches of the same project are maintained: the master branch and the nist branch. Both implement several algorithms submitted to NIST, but only the master branch can be used to use digital signatures on TLS. Thus, all the focus in this thesis is given to the master branch, since the other cannot be used to integrate the added algorithms on TLS via the authors' OpenSSL fork.

As of this writing, both CRYSTALS-Dilithium and NewHope have made it to the second round of the NIST Post-Quantum Cryptography Standardization Process, which means both are eligible to be included in liboqs.

While Dilithium was still not included on the master branch of the Open Quantum Safe Project, the same was not true for NewHope. Before considering using this project

to add post-quantum algorithms to OpenSSL, NewHope was selected as the best key exchange algorithm to integrate for the reasons described in chapter 2. However, this project already included an integration of NewHope (more precisely, NewHope-Simple), making it unnecessary to add this scheme in order to test it. Therefore, in order to allow for the testing of a new algorithm, based on M-LWE (and actually based on NewHope and Kyber, which was also present in liboqs), the new key exchange algorithm that was added was ThreeBears [Ham19], which also passed to the second round of the NIST Post-Quantum Cryptography Standardization Process. The integration process is similar, since it mostly requires adding a wrapper to existing code (preferably, the code submitted to NIST).

Because of the unexpected circumstances, the algorithms actually integrated on Open Quantum Safe were Dilithium and ThreeBears. However, the process of integration is mostly independent from the algorithm itself, especially if the code already comes supporting the NIST API.

Since the source code for Dilithium and ThreeBears submitted to NIST already has an API consistent to some of the other algorithms implemented in liboqs, as was mentioned before, the integration consists mainly on creating a wrapper around the existing code (created by the authors that submitted the algorithm) to make the core functionalities accessible through the liboqs general interface. The objective in this approach is to allow any application already using liboqs to also be able to use Dilithium or ThreeBears with the same interface without having to do anything else other than recompiling the library.

This chapter aims at describing every step taken to integrate the standalone source code for Dilithium and ThreeBears submitted to NIST into liboqs. After completing the integration, any C program compiled with the liboqs library that includes the main header file for liboqs (oqs.h) should be able to use any one of the algorithms implemented, including the new, recently added ones.

All integrations, both in the liboqs library and on the OpenSSL fork, were added and compiled on Ubuntu 18.04.

4.1 The Open Quantum Safe API

As was mentioned before, liboqs provides a general interface to use several algorithms.

Whether the algorithm is a key exchange or a digital signature, the first step when starting a new context is creating a structure that holds that context throughout the entire process. Depending on the type of algorithm, that structure can be either an OQS_SIG or an OQS_KEM. Both structures contain several algorithm specific variables like key and ciphertext lengths, as well as the algorithm that is currently being implemented, its version and NIST claimed security level. Both structures also contain three pointers to functions which are specific to

the algorithm being used. OQS_SIG contains pointers to keypair, sign and verify functions, and OQS_KEM has pointers to keypair, encaps and decaps functions.

After initializing a context OQS_SIG or OQS_KEM with the OQS_SIG_new or OQS_KEM_new functions respectively, both the private and public keys can be created by calling the keypair function inside the structure, or by using the OQS_SIG_keypair or OQS_KEM_keypair functions passing the corresponding structure as a parameter. The sign, verify, encaps and decaps functions can be used in very much the same way. Each function returns a value that informs of its success: 0 for successful, -1 for unsuccessful and 50 for error related to external libraries like OpenSSL. All values are part of an enum structure called OQS_STATUS.

When trying to use liboqs to experiment with cryptographic functions, the interface described above is sufficient to try different algorithms and test their overall performance and viability. There are other functions that are important to consider, especially when trying to use this library to add post-quantum cryptography to existing applications, such as OQS_MEM_secure_free and OQS_MEM_insecure_free, to free secret and non-secret data respectively, so that secret information doesn't stay in memory any longer than it needs to, and to prevent memory leaking. There are also standalone implementations of both AES and SHA3 algorithms, that can be used by an external application or by the schemes implemented in the library, instead of using external libraries, such as OpenSSL, to implement the same functionalities.

4.2 SOURCE CODE

Source code for any algorithm implemented in liboqs can be found under the *src* folder in the root directory of the project (which can be downloaded from the author's repository[1]). Inside this folder there are four other folders, each containing implementations for a specific type of algorithm. The *common* folder contains memory freeing functions and random generators, while the *crypto* folder contains the AES and SHA3 implementations (SHA3 includes general fixed length output functions like SHA3-256 but also the SHAKE functions that produce variable length outputs). While worthy of mention, simply because these functions can be used without needing to use the post-quantum functionalities from this library, no modifications needed to be made to these two folders, since the post-quantum algorithms are implemented in the other two, namely *sig* and *kem*. Finally, *src* also contains a header file oqs.h which includes every main header file needed to use the library. When an application uses the liboqs library, this is the only file that needs to be included with the include directive, since every other header will be included as well.

The *sig* directory, as the name itself hints, is where every post-quantum digital signature algorithm is implemented. Therefore, the source code for any new signature scheme must

1 https://github.com/open-quantum-safe/liboqs

be added under this directory, as well as the wrapper that provides a common interface with every algorithm already present.

Similarly, the *kem* directory contains the code for every key exchange scheme. To add a new one, the source code and wrapper must be included under this directory.

Initially, each of these folders contained multiple sub-folders, one for each algorithm implemented. Since this integration was accomplished in the master branch of the Open Quantum Safe Project, there were two algorithms implemented in *sig* and five in *kem*. Besides these sub-folders, there are also two source code files under both *sig* and *kem*: sig.c, sig.h, kem.c and kem.h. The sig.h header file contains a number of very important API definitions, such as the struct OQS_SIG, already briefly introduced in section 4.1, the function OQS_SIG_new which initializes the structure with the given algorithm (and also OQS_SIG_free, which frees memory allocated for OQS_SIG), the general functions for key generation, signing and verifying messages and also definitions of constants containing every algorithm's name, as well as the number of algorithms supported and which algorithm should be used by default, in case none is specified. The sig.c file implements the functions declared in sig.h. The source files kem.c and kem.h are analogous.

The source files described contain general functions and variable declarations that serve as a wrapper to the individual implementation of each algorithm. The actual implementation code for each algorithm is organized under a dedicated folder. For instance, the folder *qtesla* under *src/sig* contains all code related to the actual implementation of the qTesla algorithm. So, in order to add a new algorithm, the standard procedure would be to create a new folder for that algorithm and include all source code inside. After having the source code correctly implemented, some changes would have to be made in sig.c and sig.h (or kem.c and kem.h) to include the new algorithm in the general API functions, allowing its use in an external application to be the exactly the same as with any other algorithm in the library.

4.3 ALGORITHM INTEGRATION

To integrate an external algorithm, implemented in C, in liboqs, as was mentioned in the previous section, it is first necessary to create a new folder under its respective sub-folder. For a digital signature, the sub-folder would be */src/sig*.

Because the general structure of the liboqs API is very similar between signatures and key exchange schemes, this section will describe the integration process for Dilithium. Adding the new key exchange mechanism is quite similar to adding a signature, and thus most of the steps will be identical.

For Dilithium, the sub-folder created under *src/sig* was simply named *dilithium*. Following the examples already present in liboqs, namely the qTesla integration, the *dilithium* folder contains 3 files, sig_dilithium.c, sig_dilithium.h and Makefile.am, and a sub-folder, named

external, which contains the low level code implementation. The Makefile.am file contains information about the files needed to build the project with Dilithium, and is used to create a Makefile using **automake**.

When invoking the OQS_SIG_new function to create a new OQS_SIG context, internally the function is comparing the name of the algorithm provided as a parameter to any known algorithm in the library. If it finds a match, it will then invoke a function specific to that algorithm that creates an OQS_SIG context with all variables initialized, such as the name, the specific key and signature lengths and the keypair, sign and verify function pointers. For Dilithium, the function's name is OQS_SIG_DILITHIUM_new, and is defined in sig_dilithium.c, analogously to other signature algorithms. It allocates an OQS_SIG structure and initializes it with Dilithium's parameters, returning it in the end to the invoking function (usually OQS_SIG_new).

The sig_dilithium.h file, which is included by sig_dilithium.c, contains macro definitions for the length of both keys and the signature, values that are needed to create the OQS_SIG struct. It also contains the declaration of OQS_SIG_DILITHIUM_new and the Dilithium keypair, sign and verify functions, which follow the naming convention (in liboqs) OQS_SIG_X_Y, where X is the name of the algorithm and Y the function's name (keypair, sign and verify). So, for example, in case the implemented algorithm is Dilithium, the key generation function is OQS_SIG_DILITHIUM_keypair.

As was previously mentioned, the algorithm's three core functionalities (key generation, signing and verifying), as well as every auxiliary function, are implemented inside the *external* folder. The source code inside can be arranged in multiple ways, since there isn't a mandatory structure as to how functions need to be named or how variables need to be declared, except when dealing with the return values and parameters of the three core functions already defined in sig_dilithium.h. Generally, when trying to import an already existing algorithm, as was the case with Dilithium, there shouldn't be too many alterations that need to take place in order to correctly integrate the algorithm on liboqs, as long as the keypair, sign and verify functions follow the correct structure.

With the Dilithium integration, since the implementation used was that which was submitted by the authors to NIST, the three functions already had an interface very similar to the interface from other signature algorithms in liboqs, which use the interface from NIST submissions[2].

Inside the *external* folder, there are three main files that implement the core functionalities: DILITHIUM.c, DILITHIUM.h and DILITHIUM_api.c (names follow the naming process of other algorithms, such as qTesla). The DILITHIUM.h file declares the same three functions declared in sig_dilithium.h, and is included by DILITHIUM.c, which then implements said functions. It

2 There was only one difference, structurally, that had to be changed to allow the inclusion of Dilithium on TLS, which was the fact that the signature variable also contained the concatenated message, therefore always needing to be allocated with the static signature length parameter and an increased size for the message.

is worth mentioning that these three functions, which are the functions that will be called when using the OQS_SIG structure, still follow the naming convention from liboqs, possess the same parameters, and return the same values, no matter which algorithm is implemented.

Any one of these functions serve only as a sort of wrapper to the actual algorithm-specific code. The DILITHIUM.c file then serves as a connection between this interface and the actual functions that implement the algorithm. Figure 5 shows how this connection was implemented in liboqs, following the example from the qTesla implementation.

```
OQS_API OQS_STATUS OQS_SIG_DILITHIUM_keypair(unsigned char *pk, unsigned char *sk) {
    return crypto_sign_keypair(pk, sk);
}

OQS_API OQS_STATUS OQS_SIG_DILITHIUM_sign(unsigned char *sm, unsigned long long *smlen,
    return crypto_sign(sm, smlen, m, mlen, sk);
}

OQS_API OQS_STATUS OQS_SIG_DILITHIUM_verify(unsigned char *m, unsigned long long mlen,
    return crypto_verify(m, &mlen, sm, smlen, pk);
}
```

Figure 5: Wrapper code for Dilithium core functionalities.

The crypto_sign_keypair, crypto_sign and crypto_verify functions contain the actual code implementation for each core functionality. Thus, they are algorithm-specific. The implementation for these three functions is located in DILITHIUM_api.c.

For the integration of an algorithm to work properly with the established interface in liboqs, it is only necessary to follow the interface up to the point shown in picture 5, i.e., to the point where the bridge between the low level implementation and the interface for a certain algorithm is established. Ways to organize the low level code may vary (assuming the organization is properly done and doesn't impact security), since any programmer using the liboqs API will probably not have to worry about low level implementations at all, similarly to what happens with any programmer that uses the OpenSSL libcrypto library to access cryptographic primitives.

For Dilithium specifically, the core functions are implemented in DILITHIUM_api.c, and auxiliary functions, such as polynomial multiplication, random generators and other variable definitions (such as the definition of a polynomial type) all are implemented in specific, separate source files. Almost all of the source code's structure was maintained when integrating Dilithium's standalone implementation, with only a few modifications needed to properly connect it with the liboqs interface and with OpenSSL. One important modification was changing the return values from crypto_sign_keypair, crypto_sign and crypto_verify. As was mentioned in 4.1, even though in practical terms all keypair, sign and verify functions on liboqs return integer values of 0 or -1 (and exceptionally 50 when an external library error occurs), these belong to an enum type called OQS_STATUS that maps them, respectively,

to the names OQS_SUCCESS, OQS_ERROR and OQS_EXTERNAL_LIB_ERROR_OPENSSL. To maintain consistency, the low level functions return values are all of type OQS_STATUS, which means the possible return values are 0, -1 or 50. This return value is carried through all functions in the API and can be obtained by the programmer to check for the success or failure of the signature process. Another main modification applied to the source code from the standalone implementation was making the three core functions static, as is done in the other signature algorithms in liboqs, which is a necessary step to avoid conflicting function names, since each signature algorithm implementation can, and in fact contains, functions with the same names. The function crypto_sign, for example, exists in both DILITHIUM_api.c and QTESLAI_api.c. Making these functions static avoids conflicts between them, while allowing for the same name to be used for consistency of implementation.

As was mentioned, there is no practical opposition to organizing auxiliary code in different manners, but it is important to note that if functions or declarations in general are not static, then it becomes necessary to assure that declarations and function names do not conflict with any other that exists in liboqs that is also not static.

The last file in the *dilithium* directory, briefly mentioned before, is Makefile.am, which is a file used to automatically generate the Makefile to compile the added source code into the liboqs library. This file specifies the source code files necessary to build the new algorithm into the liboqs library, as well as some flags that are to be used. In this concrete case, the Dilithium Makefile.am file is extremely simple, following the example from qTesla. Figure 6 shows the Dilithium Makefile.am file.

```
AUTOMAKE_OPTIONS = foreign
noinst_LTLIBRARIES = libdilithium.la

libdilithium_la_SOURCES = sig_dilithium.c external/DILITHIUM.c
libdilithium_la_CFLAGS = $(AM_CFLAGS) -Iexternal
```

Figure 6: Contents from Makefile.am under *src/sig/dilithium*.

This file will be used by **automake** to generate Makefile.in, which in turn will be used by *configure* to generate a Makefile. The process is automatic, and thus adding a new algorithm only involves creating the Makefile.am file.

One important aspect to note is that there are only two source files represented in Makefile.am, the reason being that DILITHIUM.c includes the source file DILITHIUM_api.c and all auxiliary files directly. This approach allows the use of static auxiliary functions, and closely follows the approach with qTesla[3].

3 With certain key exchange schemes on liboqs, the authors employ a different way of having functions with the same name without being static, by renaming them for each algorithm using macros. That approach, however, is not mandatory, and thus the ThreeBears integration also uses static functions to avoid function naming conflicts.

Right up to this point, every modification or addition documented has been related to files specific to Dilithium, inside the *dilithium* folder. However, as was mentioned before in 4.2, there are still some files outside of the actual implementation that need to be modified, such as sig.c and sig.h.

The sig.h file defines a macro for the name of each algorithm implemented in liboqs (a set including Dilithium now). So, to give Dilithium a name to be recognized inside the library, a new macro needs to be defined for this algorithm. There is also a macro that defines how many signature algorithms there are, which is important, for instance, for the function OQS_SIG_alg_identifier, which returns the name of a signature algorithm given an integer number. Therefore, that number should be incremented for each signature scheme added to liboqs. The last modification needed is adding the include directive to include the sig_dilithium.h file.

Modifications to sig.c follow the same line. First of all, it is necessary to add Dilithium to OQS_SIG_alg_identifier, since there is now a new unique algorithm. It is also necessary to alter the OQS_SIG_new function. When receiving a certain algorithm name, the function will try to compare that name with every macro defined in sig.h (non-case sensitive), trying to find a match. If a match is found, then that algorithm's specific function for initializing OQS_SIG will be invoked. For Dilithium, that function is defined in sig_dilithium.h and implemented in sig_dilithium.c. So, in order to be able to initialize an OQS_SIG context with OQS_SIG_new, there needs to be a new condition trying to match the name provided as a parameter and Dilithium's macro defined in sig.h. In case of success, the proper function to initialize OQS_SIG with Dilithium will be called and the structure will be returned.

As far as modifications to the source code to include Dilithium go, there isn't anything else that needs to be done. However, for the purpose of this project, there were other small modifications to the source code that were implemented, to aid the integration with Python's **cryptography** package. Those will be addressed in more detail in section 4.4.

The final changes to Open Quantum Safe that need to be made to add Dilithium take place outside the *src* folder that, so far, has been the only folder described in more detail. However, there are other files that need to be altered to complete the integration.

In no particular order, the first file is features.m4, under the *config* directory. This file is responsible for defining which algorithms will be activated at compile time. To activate Dilithium, there are a couple of alterations that need to take place. Figure 7 shows the first modification that must be added.

```
AC_SUBST(SRCDIR)
AC_CONFIG_FILES([Makefile
                 src/common/Makefile
                 src/kem/Makefile
                 src/crypto/sha3/Makefile
                 src/crypto/aes/Makefile
                 src/sig/Makefile
                 src/sig/picnic/Makefile
                 src/sig/qtesla/Makefile
                 src/sig/dilithium/Makefile
                 src/kem/bike/Makefile
                 src/kem/frodokem/Makefile
                 src/kem/sike/Makefile
                 src/kem/newhopenist/Makefile
                 src/kem/kyber/Makefile
                 tests/Makefile
                 ])
```

Figure 9: Line added to configure.ac.

```
ARG_DISBL_SET_WRAP([sig-picnic], [sig_picnic],
                   [ENABLE_SIG_PICNIC], [src/sig/picnic])
ARG_DISBL_SET_WRAP([sig-qtesla], [sig_qtesla],
                   [ENABLE_SIG_QTESLA], [src/sig/qtesla])
ARG_DISBL_SET_WRAP([sig-dilithium], [sig_dilithium],
                   [ENABLE_SIG_DILITHIUM], [src/sig/dilithium])
```

Figure 7: First modification in features.m4.

The second modification is presented in figure 8.

```
AM_COND_IF([ENABLE_SIG_QTESLA], [
  AC_DEFINE(OQS_ENABLE_SIG_qTESLA_I,         1, "Define to 1 when qTESLA-I enabled")
  AC_DEFINE(OQS_ENABLE_SIG_qTESLA_III_size,  1, "Define to 1 when qTESLA-III-size enabled")
  AC_DEFINE(OQS_ENABLE_SIG_qTESLA_III_speed, 1, "Define to 1 when qTESLA-III-speed enabled")
])

AM_COND_IF([ENABLE_SIG_DILITHIUM], [
  AC_DEFINE(OQS_ENABLE_SIG_DILITHIUM, 1, "Define to 1 when DILITHIUM enabled")
])
```

Figure 8: Second modification in features.m4.

As can be seen, the necessary changes closely follow the example of previous schemes already present in liboqs.

The second file that needs to be altered is configure.ac, which can be found in the root directory of the project. There is only one line that must be added here, which also closely follows all other algorithms, and which specifies where the Makefile for Dilithium can be found. Figure 9 shows the added line.

The final modification needed to compile the library is in the Makefile.am file, located in the root of the project. There are three changes to be applied here. The first one is adding the location of the libdilithium.la, inside the *dilithium* directory. The second is adding the sig_dilithium.h header file to the list of headers to install and the third and last change is adding an instruction to copy the sig_dilithium.h header file to *include/oqs* (the directory from where header files will be included when using the API). Figures 10, 11 and 12 show these alterations.

```
if ENABLE_SIG_PICNIC
liboqs_la_LIBADD += src/sig/picnic/libpicnic_i.la
endif
if ENABLE_SIG_QTESLA
liboqs_la_LIBADD += src/sig/qtesla/libqtesla.la
endif
if ENABLE_SIG_DILITHIUM
liboqs_la_LIBADD += src/sig/dilithium/libdilithium.la
endif
```

Figure 10: Alteration to add libdilithium.la location.

```
installheaderdir=$(includedir)/oqs
installheader_HEADERS= src/oqs.h \
    src/oqsconfig.h \
    src/common/common.h \
    src/common/rand.h \
    src/crypto/aes/aes.h \
    src/crypto/sha3/sha3.h \
    src/kem/kem.h \
    src/kem/bike/kem_bike.h \
    src/kem/frodokem/kem_frodokem.h \
    src/kem/newhopenist/kem_newhopenist.h \
    src/kem/kyber/kem_kyber.h \
    src/kem/sike/kem_sike.h \
    src/sig/sig.h \
    src/sig/picnic/sig_picnic.h \
    src/sig/qtesla/sig_qtesla.h \
    src/sig/dilithium/sig_dilithium.h
```

Figure 11: Alteration to add header file.

```
links: oqsconfigh
    $(MKDIR_P) include/oqs
    cp -f src/oqs.h include/oqs
    cp -f src/oqsconfig.h include/oqs
    cp -f src/common/common.h include/oqs
    cp -f src/common/rand.h include/oqs
    cp -f src/crypto/aes/aes.h include/oqs
    cp -f src/crypto/sha3/sha3.h include/oqs
    cp -f src/kem/kem.h include/oqs
    cp -f src/kem/bike/kem_bike.h include/oqs
    cp -f src/kem/frodokem/kem_frodokem.h include/oqs
    cp -f src/kem/sike/kem_sike.h include/oqs
    cp -f src/kem/newhopenist/kem_newhopenist.h include/oqs
    cp -f src/kem/kyber/kem_kyber.h include/oqs
    cp -f src/sig/sig.h include/oqs
    cp -f src/sig/picnic/sig_picnic.h include/oqs
    cp -f src/sig/qtesla/sig_qtesla.h include/oqs
    cp -f src/sig/dilithium/sig_dilithium.h include/oqs
```

Figure 12: Alteration to copy header file to *include* directory.

After applying every modification needed to integrate Dilithium, the last step is simply to build the project using the commands from the README.md file included in the root directory of the project.

After building the project, any C program which included oqs.h and is compiled with the *-loqs* flag should be able to use the API from liboqs, including any new algorithm that was correctly integrated.

By following the build commands from README.md, the liboqs library is installed as a general library to be used in various applications. However, the authors provide in the README.md file in the separate OpenSSL fork project a slightly different list of commands that aim at installing the liboqs in a sub-directory inside the OpenSSL source code diretory. Both commands are shown in figures 13 and 14.

```
autoreconf -i
./configure
make -j
make install
```

Figure 13: Standard commands to build liboqs library.

```
autoreconf -i
./configure --prefix=../oqs-openssl/oqs --enable-shared=no
--enable-openssl --with-openssl-dir=/usr
make -j
make install
```

Figure 14: Commands to build liboqs library as a sub-folder in OpenSSL.

Since the main purpose of integrating Dilithium in liboqs is to use it on OpenSSL, the commands to use would be those of figure 14.

Because of the way the liboqs works, each algorithm integrated has a fixed parameter set, i.e., fixed key and signature lengths, fixed polynomial degrees and other parameters that describe the scheme. The Dilithium "version" that was first integrated was the recommended[4] version. However, in order to test how different different parameters affect the scheme's efficiency, the version with the strongest parameter set was also included. Including a new version with different parameters is similar to including a new algorithm, except that the file implementing the low level code (DILITHIUM_api.c) can be the same, since it essentially contains the algorithm specific implementation of the keypair, sign and verify functions, which doesn't change for each parameter set.

4.4 INTEGRATION ON PYTHON'S CRYPTOGRAPHY PACKAGE

Integrating a new algorithm on liboqs means it will from then on be available for usage as part of the liboqs API, provided the addition was accomplished correctly.

Just as a C program can make use of OpenSSL's libcrypto to use a number of different cryptographic primitives by including the *-lcrypto* flag when compiling the source code, the same can be done using the *-loqs* flag, which links the library to the program that uses the API. Because Python's **cryptography** package uses external libraries to implement its own functionalities, using what is called as a "backend", it is entirely possible to include in the package new functionalities that come from a new external library.

While in previous versions there was support for multiple backends, currently the **cryptography** package only supports the OpenSSL backend. Essentially, it means that every cryptographic algorithm supported on **cryptography** is actually implemented in the libcrypto library.

Because the main purpose of integrating the liboqs algorithms on **cryptography** is testing whether or not it's possible to use post-quantum algorithms with comparable simplicity and consistency to the already implemented classical algorithms, the integration of liboqs with **cryptography** was accomplished by including the liboqs functionalities in the OpenSSL backend, even though they come from a different library which, by itself, does not belong to OpenSSL. Should support for multiple backends still exist, a new backend for liboqs functionalities could be created, but in practice, to include basic signature and key agreement mechanisms, not much diference would result from it.

More complex functionalities, like certificate generation, were not integrated, because there would have to be several modifications to the OpenSSL source code (since the certificate generation code comes from OpenSSL), modifications that were not added by the authors of the Open Quantum Safe Project, even with all the modifications already added that are necessary to use liboqs with OpenSSL. Therefore, only public key cryptography

4 In the original paper [Duc+17a]

functionalities contained within liboqs alone were added, namely, generating private and public keys, signing and verifying functions for each algorithm and conducting a key agreement mechanism, although there was some effort to maintain the interface as consistent as possible with the interface present in other algorithms, such as DSA.

When using DSA to sign a message, the module must first be imported from **cryptography**, and then it's used to generate a private key object dependent on the chosen parameter set. The private key then is used to obtain the public key and to sign the message. Finally, the public key verifies the signature produced. Figure 15 shows a simple example of a signature process using **cryptography**.

```
from __future__ import print_function
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.exceptions import InvalidSignature

import sys


def main():

        try:
                private_key = dsa.generate_private_key(2048, default_backend())

                public_key = private_key.public_key()

                data = "This is the message to be signed."

                signature = private_key.sign(data, hashes.SHA256())

                public_key.verify(signature, data, hashes.SHA256())

        except InvalidSignature:
                print("Message verification failed!")
                return

        print("Verified successfully!")
```

Figure 15: Simple example of a signature process using DSA on **cryptography**.

To sign a message using Dilithium, for instance, the only necessary changes in the example above are in the imported module, which in this case is *oqs_sig*, and the parameter for the *generate_private_key* function, which instead of receiving the key size must receive the algorithm to be used. The same example, but now using the new liboqs algorithm Dilithium, can be seen in figure 16.

```
from __future__ import print_function
from cryptography.hazmat.primitives.asymmetric import oqs_sig
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.exceptions import InvalidSignature

import sys


def main():

        try:
                private_key = oqs_sig.generate_private_key("dilithium", default_backend())

                public_key = private_key.public_key()

                data = "This is the message to be signed."

                signature = private_key.sign(data, hashes.SHA256())

                public_key.verify(signature, data, hashes.SHA256())

        except InvalidSignature:
                print("Message verification failed!")
                return

        print("Verified successfully!")
```

Figure 16: Simple example of a signature process using Dilithium from liboqs on **cryptography**.

The **cryptography** package uses an interface called CFFI (C Foreign Language Interface), which essentially allows a Python program to call C functions in a given library. It is therefore essential that every algorithm added to **cryptography** then is "added" to this interface.

Without entering into too much detail, a Python program can include a module FFI which will then allow the creation of an ffi object. That object contains some functions to add C definitions of functions, types and includes that are needed to use the defined functions. For liboqs, for instance, it's necessary to create a file with the declarations of every C function that must be used to create keys, sign and verify, and the analogous functions for key agreement mechanisms. The contents of this file will be used, combined with the contents of all other files describing the C functions for other algorithms (like DSA), to "register" which C functions and types are available for calling in the future, using the function *cdef* in the *ffi* object and, afterwards, the *set_source* function (also from *ffi*), which makes the actual code accessible.

Because the code to include libraries and C definitions on **cryptography** is already included, adding a new library only requires modifying a function that determines which libraries will be used and creating a file with declarations for every function needed, as well as new variable types and header files that need to be included. The *ffi* object that bridges communications between Python and the C code is already included in the OpenSSL backend, and thus if added correctly, every liboqs function becomes available to be called inside the OpenSSL backend.

Because the integration itself involves multiple steps, appendix A contains a full guide on how to accomplish the integration of liboqs on **cryptography**, which allows the usage of the *oqs_sig* or *oqs_kem* modules as shown in figure 16.

5

INTEGRATION ON THE TLS PROTOCOL

The Open Quantum Safe Project , as is mentioned in [SM16], was created to allow experimentation with post-quantum cryptographic algorithms and examine their real-life usage by including them in various applications and protocols that are widely used and well established. The OpenSSL project is one of those applications, which implements the TLS protocol, where the authors created a fork from the original project to include functionalities from liboqs. The "new" project includes every functionality from the original OpenSSL (for the same version) but also includes additional code to be able to use the liboqs API and, through that API, access a number of different algorithms that aren't directly integrated on OpenSSL.

In fact, the standard approach to include new algorithms involves first adding the low level code for a specific algorithm to the sub-directory *crypto* in the OpenSSL project root directory (and ideally to the EVP layer, a high level interface for cryptographic algorithms) and then to the TLS/SSL layer [SM16][Bos+14][BWB18]. By using the Open Quantum Safe fork from OpenSSL, algorithms do not have to be added to the *crypto* folder, since their implementation comes from an external library (liboqs). There are still some modifications that take place in this folder, but nothing comparable to adding the whole low level implementation that was integrated in liboqs.

The modifications created by the authors also make it easier to add new algorithms to OpenSSL, provided those same algorithms were already integrated in the liboqs library. In fact, adding a new algorithm once implemented is fairly straightforward, since it basically involves adding new lines to existing files and functions following the example of previously added algorithms. The authors marked every spot where new code should be added to include new algorithms, both in respect to key exchange schemes and digital signatures. A simple **grep** search for the keywords ADD_MORE_OQS_SIG_HERE outputs every location where new addition need to take place to correctly add a new digital signature, and doing the same for ADD_MORE_OQS_KEM_HERE outputs the locations to add new key exchange mechanisms.

Since the Open Quantum Safe Project is meant to test and experiment new proposed post-quantum algorithms, which are fairly new compared to classical, more scrutinized ones, the authors in [SM16] recommend that these new algorithms be used in a hybrid

form[1], i.e., in conjunction with a classical algorithm (similarly to what Google conducted with its post-quantum experiment [Bra16; Lan16]) to offer post-quantum security and at the same time relying on a classical algorithm in case eventual vulnerabilities are found in the post-quantum scheme used. Therefore, the OpenSSL fork also contains the option to use algorithms in this hybrid state, by using, for example, RSA or ECDSA signature schemes with qTesla or Dilithium. Furthermore, there is no need to add any new code to the liboqs library in order to be able to use the new post-quantum algorithm in a hybrid form, since the only alterations necessary are all on the OpenSSL side.

Using this modified version of OpenSSL, as was mentioned before, allows the usage of every function available in the original project for the same version. Provided that two systems communicating with TLS via OpenSSL both possess the new libraries, there shouldn't be any problem establishing communications using any new algorithm added through liboqs. Creating certificates for signature mechanisms is also possible, and can be done using the same methods used for any other classical algorithm, as well as the hybrid formats.

5.1 Adding algorithm in configuration files

As was mentioned before, adding a new algorithm involves, for the most part, adding code following the example from other post-quantum algorithms in the project. The simpler addition of new algorithms to this modified version of OpenSSL, as opposed to the complex task of adding the same algorithms to the original project, which requires modifying possibly thousands of lines and more than 20 files, is one of the great advantages introduced with the Open Quantum Safe Project , and one of its core goals [SM16].

Following the instructions in the README file in the project's root directory, the first step to add a new signature algorithm from liboqs is altering two files in *crypto/objects*: objects.txt and obj_mac.num. Figures 17 and 18 show the added lines for both files necessary to include Dilithium.

1 In hybrid form, a message is signed by both algorithms and then the signatures are concatenated. To verify the resulting concatenated signature successfully, the verification process must be successful with both signatures.

```
# 1.3.6.1.4.1.311.89.2.1.1
# iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) Microsoft(311) MSRCrypto (89)
identified-organization 6 1 4 1 311 : Microsoft
Microsoft 89 2                      : MSRPQC
MSRPQC 1 1                          : picnicL1FS : picnicL1FS
MSRPQC 1 2                          : p256_picnicL1FS : p256_picnicL1FS
MSRPQC 1 3                          : rsa3072_picnicL1FS : rsa3072_picnicL1FS
MSRPQC 2 1                          : qteslaI : qteslaI
MSRPQC 2 2                          : p256_qteslaI : p256_qteslaI
MSRPQC 2 3                          : rsa3072_qteslaI : rsa3072_qteslaI
MSRPQC 2 4                          : qteslaIIIsize : qteslaIIIsize
MSRPQC 2 5                          : p384_qteslaIIIsize : p384_qteslaIIIsize
MSRPQC 2 6                          : qteslaIIIspeed : qteslaIIIspeed
MSRPQC 2 7                          : p384_qteslaIIIspeed : p384_qteslaIIIspeed
MSRPQC 3 1                          : dilithium : dilithium
MSRPQC 3 2                          : p256_dilithium : p256_dilithium
MSRPQC 3 3                          : rsa3072_dilithium : rsa3072_dilithium
```

Figure 17: Modifications to objects.txt to add Dilithium.

```
picnicL1FS      1197
p256_picnicL1FS     1198
rsa3072_picnicL1FS      1199
qteslaI     1200
p256_qteslaI        1201
rsa3072_qteslaI     1202
qteslaIIIsize       1203
p384_qteslaIIIsize      1204
qteslaIIIspeed      1205
p384_qteslaIIIspeed     1206
dilithium       1207
p256_dilithium      1208
rsa3072_dilithium       1209
```

Figure 18: Modifications to obj_mac.num to add Dilithium and define its NID.

As can be seen in these examples, the hybrid formats are treated as independent algorithms with their own NIDs. Therefore, each combination must be added to these files as an algorithm on its own. These files are used to automatically generate object related files using the command **make generate_crypto_objects**. The generated files (obj_dat.h, obj_mac.num, obj_mac.h and obj_xref.h) do not need to be altered after being generated.

Every further modification will take place in the source code directly. Following the indications from the authors, the files needed to be altered to add a new signature scheme are as follows:

- apps/s_cb.c
- crypto/asn1/standard_methods.h
- crypto/ec/oqs_meth.c
- crypto/evp/pmeth_lib.c
- crypto/include/internal/asn1_int.h
- crypto/include/internal/evp_int.h
- crypto/objects/obj_xref.txt
- crypto/x509/x509type.c
- include/openssl/evp.h
- ssl/ssl_cert_table.h
- ssl/ssl_locl.h
- ssl/t1_lib.c
- ssl/t1_trce.c

To add new key agreement mechanisms, the list is significantly shorter:

- apps/s_cb.c

. ssl/ssl_oqs_extra.h
. ssl/ssl_locl.h
. ssl/t1_lib.c
. ssl/t1_trce.c

If every modification for each new algorithm is applied correctly, and liboqs was compiled properly, following the instructions presented in chapter 4, then compiling the project should be as easy as following the instructions supplied by the authors in the README file.

5.2 ADDING A DIGITAL SIGNATURE

This section will delve deeper into each file mentioned in the subsection 5.1, separating them by folders. By default, every file or sub-folder will always be found under the directory is question, unless a full path is provided.

5.2.1 *Apps*

The only file that needs to be altered in this directory is s_cb.c. For signature schemes, the only place that needs to be modified is in the function get_sigtype, which return the name of an algorithm given its NID.

5.2.2 *Crypto*

The *crypto* directory contains files that need to be modified only if adding a new signature scheme, since as is mentioned by the authors in the README file, it was not possible to integrate key agreement in the EVP layer. Therefore, any addition to include key agreement mechanisms takes place in the TLS/SSL layer.

Under the *asn1* folder, the file standard_methods.h defines an array of variables of type EVP_PKEY_ASN1_METHOD, called standard_methods. For every new algorithm added (counting hybrid formats as individual algorithms), the address for the EVP_PKEY_ASN1_METHOD variable of that algorithm needs to be added. Declarations for EVP_PKEY_ASN1_METHOD variables can be found in *include/internal/asn1_int.h*, where the struct itself is defined. For each new algorithm there must be declared a new variable of type EVP_PKEY_ASN1_METHOD, following the existing examples. The defined variable's address must then be added to the standard_methods table in ascending order of NID.

The file oqs_meth.c, under the *ec* folder, contains a number of functions and definitions related to the liboqs, such as the definitions of a struct OQS_KEY, which contains an OQS_SIG struct (defined in the liboqs library), the public and private keys, an optional EVP_PKEY

```c
static char* get_oqs_alg_name(int openssl_nid)
{
  switch (openssl_nid)
    {
    case NID_picnicL1FS:
    case NID_p256_picnicL1FS:
    case NID_rsa3072_picnicL1FS:
      return OQS_SIG_alg_picnic_L1_FS;
    case NID_qteslaI:
    case NID_p256_qteslaI:
    case NID_rsa3072_qteslaI:
      return OQS_SIG_alg_qTESLA_I;
    case NID_qteslaIIIsize:
    case NID_p384_qteslaIIIsize:
      return OQS_SIG_alg_qTESLA_III_size;
    case NID_qteslaIIIspeed:
    case NID_p384_qteslaIIIspeed:
      return OQS_SIG_alg_qTESLA_III_speed;
    case NID_dilithium:
    case NID_p256_dilithium:
    case NID_rsa3072_dilithium:
      return OQS_SIG_alg_DILITHIUM;
    /* ADD_MORE_OQS_SIG_HERE */
    default:
      return NULL;
    }
}
```

Figure 19: Example of a function modified to include Dilithium.

variable for a classical algorithm in case the scheme to be used is hybrid and other informations like the NID of the scheme being used and its security bits.

Most modifications in oqs_meth.c have to do with functions that retrieve some information about an algorithm based on its NID. Modifications in these functions are essentially all the same, because most of them merely receive an NID from OpenSSL and then compare it with the NID of every liboqs algorithm added to OpenSSL. As an example, the Dilithium NID is 1207, defined as NID_dilithium. Every one of these functions will receive an integer number representing an NID and will then compare it to every NID defined for liboqs algorithms. To add Dilithium, its new NID must also be added to every case where it must be compared to the NID received as a parameter.

Figure 19 shows an example of a function in oqs_meth.c that receives an NID and returns the name of a post-quantum algorithm, defined in liboqs, if there is one.

As can be seen, the only addition needed is including the NID of the new algorithms added (post-quantum and hybrid) for comparison with the NID value received. Other functions where modifications are similar to this one are is_oqs_hybrid, get_classical_nid and get_oqs_nid, which respectively return the NID of the classical and post-quantum algorithms of a hybrid scheme, and get_oqs_security_bits.

Another function that needs to be modified is oqs_item_verify, which return 0 if the NID provided is not from a post-quantum or hybrid scheme, simply by comparing the NID received with the NID of every added scheme from liboqs, same as the other functions.

Finally, there are some function-like macros defined in this file that create new structures and functions that are algorithm-specific. Therefore, it is necessary to invoke these functions

```
#define DEFINE_OQS_EVP_PKEY_METHOD(ALG, NID_ALG)    \
const EVP_PKEY_METHOD ALG##_pkey_meth = {           \
    NID_ALG, EVP_PKEY_FLAG_SIGCTX_CUSTOM,           \
    0, 0, 0, 0, 0, 0,                               \
    pkey_oqs_keygen,                                \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    pkey_oqs_ctrl,                                  \
    0,                                              \
    pkey_oqs_digestsign,                            \
    pkey_oqs_digestverify                           \
};

#define DEFINE_OQS_EVP_METHODS(ALG, NID_ALG, SHORT_NAME, LONG_NAME)   \
DEFINE_OQS_ITEM_SIGN(ALG, NID_ALG)                                    \
DEFINE_OQS_SIGN_INFO_SET(ALG, NID_ALG)                                \
DEFINE_OQS_EVP_PKEY_METHOD(ALG, NID_ALG)                              \
DEFINE_OQS_EVP_PKEY_ASN1_METHOD(ALG, NID_ALG, SHORT_NAME, LONG_NAME)

DEFINE_OQS_EVP_METHODS(picnicL1FS, NID_picnicL1FS, "picnicL1FS", "OpenSSL Picnic L1 FS algorithm")
DEFINE_OQS_EVP_METHODS(qteslaI, NID_qteslaI, "qteslaI", "OpenSSL qTESLA-I algorithm")
DEFINE_OQS_EVP_METHODS(qteslaIIIsize, NID_qteslaIIIsize, "qteslaIIIsize", "OpenSSL qTESLA-III-size algorithm")
DEFINE_OQS_EVP_METHODS(qteslaIIIspeed, NID_qteslaIIIspeed, "qteslaIIIspeed", "OpenSSL qTESLA-III-speed algorithm")
DEFINE_OQS_EVP_METHODS(dilithium, NID_dilithium, "dilithium", "OpenSSL DILITHIUM algorithm")
```

Figure 20: Using function-like macros to create several algorithm-specific structures and functions, including Dilithium.

for each algorithm added. Figure 20 shows the usage of these macros for Dilithium. The process is exactly the same for the hybrid variations.

The DEFINE_OQS_EVP_METHODS macro invokes four other macros, one of which is DEFINE_OQS_EVP_PKEY_METHOD, and its definition can be seen in figure 20. The other macros follow a similar logic. The only alteration needed to include Dilithium is adding the invocation of DEFINE_OQS_EVP_METHODS for Dilithium, and for each hybrid variant included.

In a similar fashion to what happened with the standard_methods.h file under *asn1*, altering the pmeth_lib.c file involves adding new elements to a similar table also named standard_methods. The difference lies, however, in the type of the variables included in the array. In standard_methods.h, every element was of type EVP_PKEY_ASN1_METHOD, while in pmeth_lib.c the array standard_methods holds elements of type EVP_PKEY_METHOD, which are declared in evp_int.h under *include/internal*. The modifications needed in these two files are practically identical to those that were discussed with standard_methods.h and asn1_int.h. There is also a need to add algorithms in ascending order of their NID values.

Under the *objects* sub-folder there is a file that needs to be altered as well, called obj_xref.txt, which essentially maps an object ID (OID) to its corresponding cryptographic scheme. Figure 21 shows the additions for Dilithium and its hybrid variants.

The last file to be modified in the *crypto* folder is x509type.c. As is the case with most modifications, these are also straightforward, and take place in the x509_certificate_type function. This function returns, among other things, information about whether or not the certificate's public key can be used for signing. To allow signing with Dilithium and its hybrid variants, they have to be added to the algorithms allowed to sign. Figure 22 shows what must be added to the code.

```
# OQS signature schemes
picnicL1FS           undef     picnicL1FS
p256_picnicL1FS      undef     p256_picnicL1FS
rsa3072_picnicL1FS   undef     rsa3072_picnicL1FS
qteslaI              undef     qteslaI
p256_qteslaI         undef     p256_qteslaI
rsa3072_qteslaI      undef     rsa3072_qteslaI
qteslaIIIsize        undef     qteslaIIIsize
p384_qteslaIIIsize   undef     p384_qteslaIIIsize
qteslaIIIspeed       undef     qteslaIIIspeed
p384_qteslaIIIspeed  undef     p384_qteslaIIIspeed
dilithium            undef     dilithium
p256_dilithium       undef     p256_dilithium
rsa3072_dilithium    undef     rsa3072_dilithium
```

Figure 21: Adding Dilithium and its hybrid variants to obj_xref.txt.

```
    case EVP_PKEY_ED25519:
#if !defined(OQS_NIST_BRANCH)
    /* OQS sig schemes */
    case EVP_PKEY_PICNICL1FS:
    case EVP_PKEY_QTESLAI:
    case EVP_PKEY_QTESLAIIISIZE:
    case EVP_PKEY_QTESLAIIISPEED:
    case EVP_PKEY_DILITHIUM:
    /* ADD_MORE_OQS_SIG_HERE */
    case EVP_PKEY_P256_PICNICL1FS:
    case EVP_PKEY_RSA3072_PICNICL1FS:
    case EVP_PKEY_P256_QTESLAI:
    case EVP_PKEY_RSA3072_QTESLAI:
    case EVP_PKEY_P384_QTESLAIIISIZE:
    case EVP_PKEY_P384_QTESLAIIISPEED:
    case EVP_PKEY_P256_DILITHIUM:
    case EVP_PKEY_RSA3072_DILITHIUM:
    /* ADD_MORE_OQS_SIG_HERE hybrid only */
#endif
        ret = EVP_PKT_SIGN;
        break;
```

Figure 22: Adding Dilithium and its hybrid variants to x509_certificate_type, defined in x509type.c.

```
# define EVP_PKEY_PICNICL1FS NID_picnicL1FS
# define EVP_PKEY_QTESLAI NID_qteslaI
# define EVP_PKEY_QTESLAIIISIZE NID_qteslaIIIsize
# define EVP_PKEY_QTESLAIIISPEED NID_qteslaIIIspeed
# define EVP_PKEY_DILITHIUM NID_dilithium
/* ADD_MORE_OQS_SIG_HERE */
# define EVP_PKEY_P256_PICNICL1FS NID_p256_picnicL1FS
# define EVP_PKEY_RSA3072_PICNICL1FS NID_rsa3072_picnicL1FS
# define EVP_PKEY_P256_QTESLAI NID_p256_qteslaI
# define EVP_PKEY_RSA3072_QTESLAI NID_rsa3072_qteslaI
# define EVP_PKEY_P384_QTESLAIIISIZE NID_p384_qteslaIIIsize
# define EVP_PKEY_P384_QTESLAIIISPEED NID_p384_qteslaIIIspeed
# define EVP_PKEY_P256_DILITHIUM NID_p256_dilithium
# define EVP_PKEY_RSA3072_DILITHIUM NID_rsa3072_dilithium
/* ADD_MORE_OQS_SIG_HERE hybrid only*/
```

Figure 23: Adding the definition of EVP_PKEY_DILITHIUM and the hybrid variants to evp.h.

5.2.3 *Include*

There is only one file in this folder worthy of mention, and that is evp.h. This file defines a lot of EVP layer functions and variables, but there is only one definition that needs to be added for each algorithm, which is the **evp_pkey_x** variable, where **x** is the algorithm. In any case, this value is just defined as being the same as the NID for that algorithm. Figure 23 shows the example for liboqs schemes.

5.2.4 *SSL*

The file ssl_locl.h defines a large amount of variables and functions. Yet, like with the previous examples, modifications are only necessary in very few places throughout the file. The first place to add new information is where the SSL bitmasks are defined. For every algorithm added, there should be new values defined, and so for Dilithium and its hybrid variants new definitions must be created, following the example from other liboqs algorithms. The macros defined have names like SSL_ADIITHIUM.

The second place to be altered is where the SSL_PKEY_X are defined, where X is the signature algorithm. These are defined in ascending numbers, with SSL_PKEY_RSA being 0 and SSL_PKEY_ED448 being 8, making a total of 9 algorithms defined. Every addition from liboqs then increments the value by one. Before adding Dilithium there were 19 definitions. Because Dilithium and its variants account for 3 more, the final number is 22. This value is present in the definition of SSL_PKEY_NUM, which works as a "counter" for how many SSL_PKEY algorithms there are. These values in these definitions will be extremely important when altering the file ssl_cert_table.h.

The last alteration in ssl_locl.h is in the definitions of TLSEXT_SIGALG macros, which also appoint a unique number for each algorithm. These definitions are used in t1_lib.c. As with

many modifications before, new additions can just follow the example from other already added schemes.

The t1_lib.c file implements a lot of functionalities for the actual TLS/SSL protocol. There are a total of three modifications, for each new algorithm, that need to take place in this file. First an foremost, the new algorithms need to be added to the tls12_sigalgs array, holding the default signature algorithms. Adding Dilithium to this table means adding the TLSEXT_SIGALG_dilithium macro defined in ssl_locl.h, and doing the same for the hybrid formats.

The array sigalg_lookup_tbl contains elements of type SIGALG_LOOKUP, which is essentially a struct containing relevant values, such as the name, NID and extension value (TLSEXT_SIGALG) associated with a specific algorithm. Functions like tls1_lookup_sigalg use this structure to be able to find a specific algorithm, and are used in other functions like tls12_check_peer_sigalg, which checks if a signature algorithm from the other peer is consistent with the supported signature schemes. Therefore, entries for each new algorithm must be added as well, following the example of other algorithms.

Finally, in the tls1_set_cert_validity function, which validates certificates received, new code has to be added to account for any new algorithm.

Figures 24, 25 and 26 show all modifications in t1_lib.c to add Dilithium and its variants.

```
#if !defined(OQS_NIST_BRANCH)
    /* OQS sig schemes*/
    TLSEXT_SIGALG_picnicL1FS,
    TLSEXT_SIGALG_qteslaI,
    TLSEXT_SIGALG_qteslaIIIsize,
    TLSEXT_SIGALG_qteslaIIIspeed,
    TLSEXT_SIGALG_dilithium,
    /* ADD_MORE_OQS_SIG_HERE */
    /* OQS hybrid schemes*/
    TLSEXT_SIGALG_p256_picnicL1FS,
    TLSEXT_SIGALG_rsa3072_picnicL1FS,
    TLSEXT_SIGALG_p256_qteslaI,
    TLSEXT_SIGALG_rsa3072_qteslaI,
    TLSEXT_SIGALG_p384_qteslaIIIsize,
    TLSEXT_SIGALG_p384_qteslaIIIspeed,
    TLSEXT_SIGALG_p256_dilithium,
    TLSEXT_SIGALG_rsa3072_dilithium,
```

Figure 24: Adding Dilithium to the tls12_sigalg table.

```
#if !defined(OQS_NIST_BRANCH)
    /* OQS sig schemes */
    {"picnicL1FS", TLSEXT_SIGALG_picnicL1FS,
     NID_undef, -1, EVP_PKEY_PICNICL1FS, SSL_PKEY_PICNICL1FS,
     NID_undef, NID_undef},
    {"qteslaI", TLSEXT_SIGALG_qteslaI,
     NID_undef, -1, EVP_PKEY_QTESLAI, SSL_PKEY_QTESLAI,
     NID_undef, NID_undef},
    {"qteslaIIIsize", TLSEXT_SIGALG_qteslaIIIsize,
     NID_undef, -1, EVP_PKEY_QTESLAIIISIZE, SSL_PKEY_QTESLAIIISIZE,
     NID_undef, NID_undef},
    {"qteslaIIIspeed", TLSEXT_SIGALG_qteslaIIIspeed,
     NID_undef, -1, EVP_PKEY_QTESLAIIISPEED, SSL_PKEY_QTESLAIIISPEED,
     NID_undef, NID_undef},
    {"dilithium", TLSEXT_SIGALG_dilithium,
     NID_undef, -1, EVP_PKEY_DILITHIUM, SSL_PKEY_DILITHIUM,
     NID_undef, NID_undef},
```

Figure 25: Adding Dilithium to the tls1_lookup_tbl array.

```
#if !defined(OQS_NIST_BRANCH)
    /* OQS sig schemes */
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_PICNICL1FS);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_QTESLAI);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_QTESLAIIISIZE);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_QTESLAIIISPEED);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_DILITHIUM);
    /* ADD_MORE_OQS_SIG_HERE */
    /* OQS hybrid schemes */
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_P256_PICNICL1FS);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_RSA3072_PICNICL1FS);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_P256_QTESLAI);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_RSA3072_QTESLAI);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_QTESLAIIISIZE);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_QTESLAIIISPEED);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_P256_DILITHIUM);
    tls1_check_chain(s, NULL, NULL, NULL, SSL_PKEY_RSA3072_DILITHIUM);
    /* ADD_MORE_OQS_SIG_HERE hybrid only */
```

Figure 26: Adding Dilithium to the tls1_set_cert_validity function.

The last file that needs to be altered in the *SSL* folder is t1_trce.c, and the only modification needed is adding the new algorithm to the ssl_sigalg_tbl array, similarly to what happens with the tls12_sigalg array from t1_lib.c. Figure 27 shows the additions of Dilithium and the hybrid variants.

```
#if !defined(OQS_NIST_BRANCH)
    /* OQS sig schemes */
    {TLSEXT_SIGALG_picnicL1FS, "picnicL1FS"},
    {TLSEXT_SIGALG_qteslaI, "qteslaI"},
    {TLSEXT_SIGALG_qteslaIIIsize, "qteslaIIIsize"},
    {TLSEXT_SIGALG_qteslaIIIspeed, "qteslaIIIspeed"},
    {TLSEXT_SIGALG_dilithium, "dilithium"},
    /* ADD_MORE_OQS_SIG_HERE */
    /* OQS hybrid schemes */
    {TLSEXT_SIGALG_p256_picnicL1FS, "p256_picnicL1FS"},
    {TLSEXT_SIGALG_rsa3072_picnicL1FS, "rsa3072_picnicL1FS"},
    {TLSEXT_SIGALG_p256_qteslaI, "p256_qteslaI"},
    {TLSEXT_SIGALG_rsa3072_qteslaI, "rsa3072_qteslaI"},
    {TLSEXT_SIGALG_p384_qteslaIIIsize, "p256_qteslaIIIsize"},
    {TLSEXT_SIGALG_p384_qteslaIIIspeed, "rsa3072_qteslaIIIspeed"},
    {TLSEXT_SIGALG_p256_dilithium, "p256_dilithium"},
    {TLSEXT_SIGALG_rsa3072_dilithium, "rsa3072_dilithium"},
```

Figure 27: Adding Dilithium to the tls1_set_cert_validity function.

5.3 ADDING A KEY AGREEMENT MECHANISM

Adding a key agreement mechanism from liboqs, as shown above, involves altering many fewer files than adding a signature mechanism, mainly because it doesn't involve modifying anything in the *crypto* folder.

5.3.1 *Apps*

As is the case with the addition of a digital signature, when it comes to altering the *apps* folder, adding a new key agreement only requires modifying s_cb.c, more precisely, the function OQS_CURVE_ID_NAME_STR, which receives an ID and returns the scheme's name. The function simply analyses the ID received and compares it to specific values to see if it matches any. The name returned, in case of success, will just be used to print information about which algorithm was used to establish the server temporary key. Adding a new scheme requires the addition of a new, unique value to match the ID value received, following the example from previously added algorithms.

5.3.2 *SSL*

There are four files in the *ssl* folder which require modifications, three of which also need to be altered to add a digital signature.

The file ssl_oqs_extra.h has only one function, which is oqs_nid_from_string, a function that returns the key agreement scheme's NID given its name. Adding a new scheme is as simple as adding a new condition that tries to match the received name with the algorithm's and, in case of success, returning that new algorithm's NID.

The first modification in ssl_locl.h has to do with defining the schemes' NID values. The authors comment that the NID values were added here because otherwise the crypto layer would have to be altered. As was the case with digital signatures, adding a new value must take into account that it must be unique and, preferably, incremental to previous schemes' NID values as well.

The new scheme must also be added to the function-like macro OQS_KEM_CURVEID, in case the algorithm was added to the liboqs master branch or OQS_KEM_CURVE_ID_NIST_BRANCH, in case the scheme was added to the liboqs nist branch. In any case, the functions return the curve ID (the same ID that is received as a parameter in OQS_CURVE_ID_NAME_STR from s_cb.c) given an NID of a key agreement scheme. If none is found, it returns 0. The exact same process must be followed if the scheme added is a hybrid scheme, in which case it must be added to OQS_KEM_HYBRID_CURVEID or OQS_KEM_HYBRID_CURVEID_NIST_BRANCH.

The opposite function-like macros, that is, functions that return curve IDs given an NID, are also defined. OQS_KEM_NID and OQS_KEM_NID_NIST_BRANCH follow the same logic from their counterparts, and adding an algorithm means comparing the curve IDs received and returning the corresponding NID. However, unlike their counterparts, these functions do not have a hybrid variant, since the NID returned for a post-quantum scheme and its hybrid variant is the same, i.e., the NID of the post-quantum scheme.

There also function-like macros that check if a given curve ID is from a liboqs scheme. For example, IS_OQS_KEM_CURVEID and IS_OQS_KEM_HYBRID_CURVEID simply check if the ID is between a certain range of curve IDs. If a new algorithm is added, hybrid or not, the range needs to be altered to make sure the new algorithm gets correctly categorized.

Finally, the function-like macros OQS_ALG_NAME and OQS_ALG_NAME_NIST_BRANCH return the key agreement scheme's name given its NID. The name returned is the name defined in liboqs. The logic between the NIST branch variation is the same from the other examples above.

All modifications in t1_lib.c, regarding key agreement mechanisms, occur in three distinct arrays: oqs_nid_list, oqs_hybrid_nid_list and eccurves_default. The first two arrays have elements of type TLS_GROUP_INFO, which contains the NID of the scheme, the security bits and a flag value to identify its type. Adding a new algorithm is as simple as adding a new element using its NID and bits of security, since the flag used for every element is the same.

The array eccurves_default contains all curve IDs to be used, and therefore any new scheme must have its own curve ID added as well.

Finally, the t1_trce.c file has but one modification necessary, in the array ssl_groups_tbl, which is an array that maps a curve ID to a scheme's name, so it naturally follows that adding a new algorithm means adding a new entry to this table following the example from other schemes.

6

RESULTS

The classical algorithms that have now seen extensive use throughout the years have been thoroughly put to test in real-world scenarios, allowing to establish secure communications through the internet and other mediums. However, since they might eventually become insecure, it's important that the post-quantum algorithms that could replace them be as efficient as possible, and allow for a transition as seamless as possible.

Certain cryptographic operations, such as certificate generation (which involves generating key pairs and signatures) and establishing TLS connections between clients and server are crucial to safely communicate through the internet, and so are some of the most important aspects to take into account when replacing classical algorithms.

All tests using the algorithms included in the liboqs library were conducted on Ubuntu 18.04. Certificate generation was accomplished on a virtual machine with 6GB of RAM and with an Intel Core i7-4790 processor. For TLS connections, tests took place on the same virtual machine, for the server, and on a machine with a Core i5-7200U processor, with 8GB of RAM, for the client.

6.1 OPENSSL TOOLKIT

OpenSSL, as was previously mentioned, is a cryptographic toolkit, which implements a general purpose library *libcrypto*, composed of several cryptographic functions, and a library *libssl* implementing the SSL/TLS protocol, using the algorithms in *libcrypto*. Using these libraries, it is possible to write client and server applications which communicate using the TLS protocol. However, OpenSSL already offers two server and client applications that contain some functionalities aiming at testing connections with different algorithms. Even though there are some limitations to the algorithms integrated in the Open Quantum Safe Project, such as not adding the key exchange algorithms to the EVP layer, the modifications are compatible with these two test applications, making it easier to test and compare post-quantum algorithms with their classical counterparts.

The two applications, **s_server** and **s_client** implement, respectively, a test server and a test client application that can be tweaked using command line parameters. Using these applications, it's possible to create a TLS connection between two distinct computers and test different algorithm combinations (post-quantum, classical and hybrid variations) to analyse how they compare against each other.

Certificate generation, which is accomplished using the **req** application also available directly on OpenSSL, is compatible with the Open Quantum Safe modifications too, allowing to create certificates for post-quantum algorithms and its hybrid variants.

After building OpenSSL, it is straightforward to start these two applications, using the command *openssl s_server* or *openssl s_client* followed by whichever flags are needed to tweak parameters about the connection to establish.

Provided a private key and a certificate have already been generated, the server must load them using the *-cert* and *-key* flags. Other flags might also be important, depending on the actual usage. Figure 28 shows the command that was used to start **s_server**. Likewise, figure 29 shows the corresponding command used to launch **s_client**.

```
openssl s_server -cert server_CA.crt -key server_CA.key -WWW -tls1_3 -accept localhost:4433
```

Figure 28: Command used to start OpenSSL's **s_server** application.

```
openssl s_client -curves <kex> -CAfile server_CA.crt -connect localhost:4433
```

Figure 29: Command used to start OpenSSL's **s_client** application.

The **s_server** flags *-cert* and *-key* are, as mentioned above, used to load the certificate (for the signature algorithm to be used) and its corresponding private key. The *-WWW* flag makes **s_server** work as a simple web server that responds to HTTP commands, the *-tls1_3* flag instructs the server to use the TLS 1.3 protocol and the *-accept* flag instructs the server in which IP address and port connections must be accepted. By default, the port number is 4433.

On the **s_client** side, the *-curves* flag specifies which algorithm will be used for key exchange. To "force" the usage of a specific algorithm, given that ultimately it is the server that chooses which one will be used, only one algorithm is specified in the command. The root CA certificate to validate the server certificate can be included with the *-CAfile* flag, which in this case uses the same certificate used by the server (another certificate can be used, with the condition that it is signed by the CA). Finally, the *-connect* flag instructs to connect to a specific IP address and port, defaulting to localhost and the 4433 port if nothing is specified.

Running the two applications with the commands above in two terminals, assuming everything is setup properly, initiates a TLS handshake with the signature and key exchange algorithms specified, producing the output shown in figure 30.

```
tgbAtScrCLeZV1bDo3WiWZToo1MwUTAdBgNVHQ4EFgQUQHAqB40nR2dN3WEzewh3
uQ+LcbQwHwYDVR0jBBgwFoAUQHAqB40nR2dN3WEzewh3uQ+LcbQwDwYDVR0TAQH/
BAUwAwEB/zAKBggqhkjOPQQDAgNJADBGAiEAgdiuXnwhQjEZC3E3iijT+Gz0MZgI
LX64zLHFehfGJmICIQCzJ3oj4PjCOn4QIjvdB1iJLabBb8KfwbU8JjdmJ/cbZw==
-----END CERTIFICATE-----
subject=CN = oqstest

issuer=CN = oqstest

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: newhope1024cca
---
SSL handshake has read 3004 bytes and written 2201 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID: 22DC7F9D1459999A6C2B489B6FF53F60389423AB51D98D1D4AB37A97483D5246
    Session-ID-ctx:
    Resumption PSK: 2EA4AB671A6C89C59674F9BF1A8685BE33E1842FA99C6F6B29973790F7349E16F
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 7200 (seconds)
    TLS session ticket:
    0000 - 24 2b c9 e1 d2 ba a3 f0-e4 55 9e 61 3d 02 57 08   $+.......U.a=.W.
```

Figure 30: Part of the resulting output from established connections between **s_server** and **s_client** using the parameters shown in figures 28 and 29.

The next two sections showcase, respectively, measurements regarding certificate generations and TLS connections. In both cases, time was measured using the Ubuntu bash command **time**, using the elapsed real time (wall clock time as described by the **man** page for the **time** command).

6.2 CERTIFICATE GENERATION

Because certificate generation involves generating key pairs and producing signatures, the time it takes to produce one can vary significantly depending on the algorithm itself. On top of that, because different algorithms have different key and signature sizes, certificates in the end can also vary in size, which is an important element to consider when using them to establish communications between a client and a server.

It is usually the case that when dealing with LWE and R-LWE based schemes, while the general speed of a scheme is on par with the most used classical algorithms, sometimes even surpassing them, the key and signature sizes are also significantly larger, raising some concerns about how much the increased size might influence the scheme's viability. Therefore, it's useful to consider the trade-off between speed and size when evaluating different algorithms, to gain a better understanding of how big the impact of a possible replacement would actually be.

Using the **req** application, it's possible to generate a certificate for a given signature algorithm with the command in figure 31.

```
openssl req -x509 -new -newkey <sig> -keyout <sig>_CA.key -out <sig>_CA.crt
-nodes -subj "/CN=oqstest CA" -days 365 -config apps/openssl.cnf
```

Figure 31: Command used to generate a root CA certificate using the chosen signature algorithm.

The command contains several important flags. The *-x509* option makes the command output a self signed certificate, instead of a certificate request, while the *-new* starts the certificate request generation process. Flags *-newkey* and *-keyout* are, respectively, related to the private key generation for a specific signature algorithm and its output to a key file. The *-out* option specifies the output file for the certificate (which for simplicity, in this case, is named after the algorithm used). Finally, the *-subj* flag selects the subject name in the request/certificate.

Additionally, there is some useful information that might be added to specify certain aspects in the output files. Using the *-nodes* flag makes the application output the private key without encrypting it, using *-days* specifies the validity of the output certificate, assuming the *-x509* flag was also used. By default, the validity is 30 days. If an external configuration file for certificate generation is needed, it can be included with the *-config* option.

To generate an ECDSA certificate, the command changes on the *-newkey* option (figure 32), but the rest remains the same.

```
openssl req -x509 -new -newkey ec:<(openssl ecparam -name <curve>) -keyout <sig>_CA.key
-out <sig>_CA.crt -nodes -subj "/CN=oqstest CA" -days 365 -config apps/openssl.cnf
```

Figure 32: Command used to generate a root CA certificate using ECDSA.

To check Dilithium's viability on the post-quantum scene, as well as other algorithms already present on OpenSSL, the time spent to generate a certificate for each algorithm was measured. In order to compare the post-quantum algorithms with their classical counterparts, certificate generation was also timed for ECDSA and RSA signature algorithms, with varying parameter sizes. Table 7 shows how much time it took to generate each certificate, on

average, for each algorithm and the size of each certificate. Every generated certificate was generated using the command from figure 32, which only generates a self-signed root CA certificate.

Algorithm	Bit Security	Security level	Generation time (ms)	Size (bytes)
Classical Algorithms				
RSA-2048	112/0	-	86.754 ± 53.882	1115
RSA-3072	128/0	-	313.918 ± 213.775	1464
RSA-4096	150/0	-	754.214 ± 475.708	1809
ECDSA p256	128/0	-	5.072 ± 0.362	570
ECDSA secp384	192/0	-	7.074 ± 0.702	652
ECDSA secp521	256/0	-	10.05 ± 0.485	753
Post-Quantum Algorithms				
Picnic-L1-FS	128/64	1	8.202 ± 0.917	45140
qTesla-I	103/96	1	5.342 ± 0.627	4304
Dilithium	141/128	2	4.902 ± 0.751	6055
Dilithium-Max	174/158	3	4.844 ± 0.986	7347
qTesla-III-speed	178/164	3	7.25 ± 12.835	8464
qTesla-III-size	180/166	3	8.426 ± 2.259	8119
Hybrid Algorithms (Classical and Post-Quantum)				
P256 Dilithium	128/128	2	5.132 ± 0.532	6250
RSA3072 Dilithium	128/128	2	319.714 ± 206.572	7123
P256 qTesla-I	103/96	1	6.354 ± 18.932	4499
RSA3072 qTesla-I	103/96	1	309.982 ± 191.058	5372

Table 7: Table showcasing time spent to generate a certificate for each signature algorithm, as well as their respective sizes in bytes. Bit security represents the classical and post-quantum security for each scheme, respectively. Generation time corresponds to the average time (and standard deviation) spent to generate a certificate, from a total of 500. Time is represented in milliseconds, rounded to 3 decimal places.

The algorithms in the table are divided into 3 categories: classical algorithms, which are the standard algorithms currently used for digital signatures, post-quantum algorithms, and hybrid variants, which contains both classical and post-quantum algorithms.

NIST describes, in its call for proposals [ST16], 5 levels of security for each scheme to be categorized with:

- L1: An attack that breaks an L1 scheme must require computational resources similar to or greater than the resources needed to accomplish a 128-bit key search on a block cipher, like AES-128.
- L2: An attack that breaks an L2 scheme must require computational resources similar to or greater than the resources needed to search for a collision on a 256-bit hash function, like SHA-256.
- L3: An attack that breaks an L3 scheme must require computational resources similar to or greater than the resources needed to accomplish a 192-bit key search on a block cipher, like AES-192.
- L4: An attack that breaks an L4 scheme must require computational resources similar to or greater than the resources needed to search for a collision on a 384-bit hash function, like SHA-384.
- L5: An attack that breaks an L5 scheme must require computational resources similar to or greater than the resources needed to accomplish a 256-bit key search on a block cipher, like AES-256.

Comparing different algorithms (for a fixed set of parameters) becomes easier and perhaps more meaningful if said algorithms have similar security , i.e., share the same security levels proposed by NIST.

From the same document, it is also recommended that the submitters focus on schemes targeting the first 3 levels of security, those ”likely to provide sufficient security for the foreseeable future” [ST16]. Dilithium, according to the authors in [Duc+17b], achieves level 3 when using the largest set of parameters (identified as ”Dilithium-Max”, the other ”version” of Dilithium included in liboqs). Algorithms targeting levels 4 and 5, while present in some submissions, are not presented in table 7.

Looking at the results, there are some observations about certain scheme's apparent viability that are relevant to point out.

As was expected, there is a clear difference between the size of certificates from classic and post-quantum algorithms, the former being significantly shorter that the latter. For instance, for a similar classical security level of around 128 bits, a Dilithium certificate has a size approximately 4 times larger than a 3072-bit RSA certificate, and over 10 times larger than a 256-bit ECDSA certificate. The qTesla-I scheme, having around 100 bits of classical security, is still nearly 3 times larger than the 3072-bit RSA certificate, and 7.5 times larger the corresponding ECDSA certificate. For its more secure variants, the differences grow even higher.

When comparing the R-LWE based post-quantum algorithms with each other, it is noteworthy that the Dilithium-Max scheme, which is level 3, compares rather favourably to both variants of the level 3 qTesla scheme. In fact, generation times were identical between both versions of Dilithium (which differ only on the parameter sets), and between these versions and qTesla-I. Generation times for qTesla-III-speed and qTesla-III-size were slightly worse, although still relatively low. Furthermore, certificates produced with Dilithium-Max are smaller than the qTesla counterparts, for the same security level.

The Picnic signature scheme, which is not LWE based scheme, was included in the testing because its signatures are significantly larger than that of any other scheme. A certificate for the L1 variant produces signatures of at most 34020 bytes, resulting in certificates around 31 times bigger than a 3072-bit RSA certificate for the same 128 bits of classical security, and 72.2 times larger than ECDSA-P256 certificates. Generation times also seem to be slightly bigger, on average, than those of the other post-quantum algorithms, all based on some variant of the LWE problem.

In order to reflect on the viability of replacing, for instance, RSA with a Dilithium or qTesla scheme, it's important to consider just how much the added certificate size will influence communications on protocols like TLS, which is essential in securing communications on the internet.

There is, however, another important aspect to consider when comparing certificates between algorithms, and that is the time it takes to generate a new certificate. Depending on the algorithm, producing a key pair for a given security parameter might become too time consuming to be viable in real-world usage, even more when considering that security parameters might increase with time, as computers become faster. Looking again at table 7, the difference between the average time necessary to generate an RSA certificate and a certificate for any of the other algorithms is clear, as it is one or two orders of magnitude slower. Furthermore, as the bit security increases from one RSA parameter set to another, time consumption seems to increase exponentially.

Certificate generation for ECDSA is much faster when compared to RSA, for every level of security, and the same is true when comparing post-quantum schemes. In fact, when comparing generation times, ECDSA seems to have a similar generation time, on average, to that of each post-quantum algorithm, for a given security level, with ECDSA certificates being slightly faster to generate. Both kinds seem to also scale relatively well with the increase of their security parameters, as opposed to RSA.

Finally, when looking at the hybrid variants, which combine two signature algorithms, it can be seen that certificate sizes are predominantly determined by the post-quantum algorithm. For example, RSA3072 with Dilithium is only approximately 1.17 times bigger than the Dilithium certificate. The same comparison between ECDSA P256 with Dilithium shows only an increased size of approximately 1.03 times the size of the Dilithium certificate. When comparing generation times, generating a hybrid certificate with these two algorithm cost almost as much time as generating a certificate just for the ECDSA part.

Given the results obtained, it seems reasonable to assess that , when it comes to generating certificates, should a post-quantum algorithm be used, the hybrid version seems to have a small additional cost on both size and generation times. Because post-quantum schemes haven't yet been scrutinized to the extent of classical, standardized algorithms, it might be an acceptable choice to use a hybrid scheme (which was in fact recommended by the authors in [SM16]), which offers the security from classical schemes and the potential security against quantum computers from the post-quantum algorithms.

6.3 TLS CONNECTIONS

While it is important to benchmark certificate generations when testing the performance of different signature algorithms, it is also important to test how efficient each scheme can be when being used to establish TLS communications between parties. On top of that, certificate generation only involves signature schemes, and not key exchange mechanisms. The latter can be tested by establishing communications with different combinations of

signature schemes and key exchange mechanisms, and measuring how much time each communication takes, on average.

To maintain consistency throughout the whole process, the commands used for establishing a communication, both with the **s_server** and **s_client** applications, were always the same, and were shown, respectively, in figures 28 and 29. There is, however, a small difference in the **s_server** command, since the certificate and private key files used were not the self-signed, CA versions. Instead, new keys and corresponding certificates were generated and then signed by the CA private key. The CA certificates were then used by the **s_client** application to verify the certificates generated for the server. Replacing the certificate to be used in the **s_server** application involves only replacing the CA certificate and private key files by the new files that were signed by the CA. Because the **s_client** application must still use the CA certificate to validate the server certificate, the command remained unchanged (aside from the localhost:4433 part, which must be changed on both commands to match the server's IP address and listening port).

The main reason to use a different, CA signed certificate for the server is to have a certificate chain (albeit a very small one) that must be verified by the client, instead of just having to verify the root CA certificate.

Table 8 shows measurements for 6 different signature algorithms, combined with 6 key exchange schemes, with a combined total of 36 types of connections.

Signature	Key Exchange	Handshake data	Time (1kB)	Time (100kB)
RSA-3072	NewHope512 CCA	2799(r)/1287(w)	74.726 ± 28.715	170.866 ± 48.989
	ECDHE P256	1744(r)/424(w)	75.178 ± 55.906	181.604 ± 52.083
	P256 - NewHope512 CCA	2872(r)/1360(w)	77.788 ± 57.507	183.236 ± 57.913
	Frodo640 w/ cSHAKE	11415(r)/9975(w)	118.16 ± 59.549	241.272 ± 100.481
	SIKE P503	2081(r)/737(w)	252.806 ± 115.91	366.18 ± 134.209
	ThreeBears Baby	2600(r)/1167(w)	67.688 ± 25.329	178.07 ± 36.738
ECDSA P256	NewHope512 CCA	1831(r)/1287(w)	66.076 ± 39.793	169.204 ± 48.196
	ECDHE P256	776(r)/424(w)	62.876 ± 18.166	160.9 ± 45.577
	P256 - NewHope512 CCA	1904(r)/1360(w)	67.728 ± 22.154	165.078 ± 61.718
	Frodo640 w/ cSHAKE	10446(r)/9975(w)	113.314 ± 32.879	207.3 ± 73.738
	SIKE P503	1112(r)/737(w)	268.46 ± 206.782	390.4 ± 150.427
	ThreeBears Baby	1631(r)/1167(w)	65.836 ± 61.752	174.6 ± 22.338
Dilithium	NewHope512 CCA	8507(r)/1287(w)	72.232 ± 23.174	161.746 ± 31.075
	ECDHE P256	7452(r)/424(w)	75.174 ± 24.433	163.46 ± 38.119
	P256 - NewHope512 CCA	8580(r)/1360(w)	71.396 ± 11.244	165.974 ± 61.794
	Frodo640 w/ cSHAKE	17123(r)/9975(w)	110.44 ± 15.744	221.07 ± 52.673
	SIKE P503	7789(r)/737(w)	289.814 ± 165.361	384.848 ± 106.55
	ThreeBears Baby	8308(r)/1167(w)	73.126 ± 28.204	182.996 ± 62.314
qTeslaI	NewHope512 CCA	5889(r)/1287(w)	69.732 ± 16.763	178.918 ± 143.372
	ECDHE P256	4834(r)/424(w)	67.788 ± 13.262	169.688 ± 45.981
	P256 - NewHope512 CCA	5962(r)/1360(w)	68.606 ± 16.534	180.078 ± 70.935
	Frodo640 w/ cSHAKE	14505(r)/9975(w)	114.556 ± 30.692	237.712 ± 78.621
	SIKE P503	5171(r)/737(w)	244.722 ± 80.138	364.742 ± 167.781
	ThreeBears Baby	5690(r)/1167(w)	77.092 ± 38.303	175.804 ± 27.318
P256-Dilithium	NewHope512 CCA	8725(r)/1287(w)	72.872 ± 13.837	163.12 ± 57.314
	ECDHE P256	7671(r)/424(w)	77.652 ± 19.735	170.042 ± 60.083
	P256 - NewHope512 CCA	8799(r)/1360(w)	79.268 ± 21.614	171.898 ± 54.951
	Frodo640 w/ cSHAKE	17341(r)/9975(w)	114.58 ± 32.949	248.164 ± 165.166
	SIKE P503	8007(r)/737(w)	264.636 ± 132.241	355.48 ± 84.775
	ThreeBears Baby	8527(r)/1167(w)	80.132 ± 66.447	170.428 ± 23.353
P256-qTeslaI	NewHope512 CCA	6109(r)/1287(w)	70.69 ± 12.105	182.306 ± 64.11
	ECDHE P256	5055(r)/424(w)	73.878 ± 20.675	156.762 ± 38.161
	P256 - NewHope512 CCA	6182(r)/1360(w)	77.02 ± 32.617	170.496 ± 53.923
	Frodo640 w/ cSHAKE	14724(r)/9975(w)	113.08 ± 19.322	221.652 ± 69.046
	SIKE P503	5391(r)/737(w)	237.956 ± 155.233	368.644 ± 93.932
	ThreeBears Baby	5911(r)/1167(w)	76.612 ± 34.43	177.718 ± 41.995

Table 8: Table showcasing time spent to establish a TLS 1.3 connection between **s_client** and **s_server**, as well as amount of data exchanged during the handshake, for different signature algorithms and key agreement mechanisms. Time is represented in milliseconds, rounded to 3 decimal places. For each combination, it was first measured how much time it took to get a 1kB file from the server, and then the same procedure was followed for a 100kB file. Every value corresponds to the average time calculated from 500 connections, for both files.

Every key exchange and the signature scheme showcased in the table provides, approximately, the same amount of security bits. Specifically, every post-quantum algorithm is a level 1 or 2 scheme (already shown in table 7), including NewHope, Frodo and ThreeBears (the Baby version is a level 1 scheme), and every classical algorithm offers around 128 bits of classical security.

Even though there was an effort in trying to compare different schemes with similar security properties, there were some exceptions. SIKE [sJA19], for instance, which is not an LWE or R-LWE based scheme, and was included just to compare the gains or losses in

efficiency with algorithms that are based on these problems (which is also the reason Picnic was included in the testing showcased in 6.2) is a level 2 scheme, while every other key exchange mechanism is level 1. In this case, the reason is that Open Quantum Safe does not include the level 1 version of SIKE in the OpenSSL integration, and thus the level 2 version was the lowest that could be used.

Dilithium is also a level 2 scheme (as is claimed by the authors in [Duc+17b]), which is being compared to level 1 schemes such as qTeslaI. However, the reason behind this comparison is that Dilithium was purposefully used with its recommended parameters (recommended in the same paper), to achieve around 128 bit security, both classical and post-quantum, which is especially useful when comparing Dilithium to classical schemes like ECDSA-P256 or RSA-3072.

The authors of the Open Quantum Safe Project, when defining the hybrid variants for the OpenSSL integration, joined schemes based on the NIST security level and security bits the schemes offered. For instance, the NewHope512 variant, which is a level 1 scheme, was grouped with ECDHE with curve P-256, which gives a security of around 128 bits. Level 3 signature schemes, such as the qTesla-III variants for example, were grouped with ECDSA with curve P-384, which provides around 192 bits of security.

As can be seen in the table, the average time for each connection type varies slightly with different signature schemes, with ECDSA P256 and qTeslaI seemingly having slightly better times, for 1kB file transfers, than the rest of the signature algorithms. When considering 100kB, Dilithium achieves average times similar to ECDSA P256, and so does the hybrid variant, P256-Dilithium. It is worthy of mention, however, that some standard deviation values are quite big, especially when comparing times for 100kB files.

When comparing key exchange mechanisms, however, clear differences between some algorithms start to emerge. Specifically, regardless of the signature algorithm used, the Frodo640 with cSHAKE scheme seems to take a significant amount of time more than every other scheme, with the exception of SIKE, which is consistently slower than Frodo640 (and with a larger difference than Frodo640's difference to any other algorithm tested). The difference is especially noteworthy when comparing times for 1kB files, while still being present with 100kB files. Because 1kB files are shorter that the 100kB counterparts, the percentage of time spent in the handshake process is larger for the former and thus, times measured with 1kB files appear to be more indicative of how much impact a certain key exchange scheme has on the overall connections.

The handshake data in each combination corresponds to the read and written bytes, respectively, shown in the output of using the **s_server** and **s_client** applications with the commands shows above. The output itself was already shown in figure 30. As expected, given the increased sizes for public keys and signatures, post-quantum schemes reveal a substantial increase in bytes read during the handshake process, with RSA-3072 and

especially ECDSA-P256 having a significantly lower amount, for the same key exchange scheme. Dilithium has the largest amount of bytes read during handshakes, with the only exception being the P256-Dilithium hybrid, which has approximately 200 bytes more. When comparing handshake bytes, it is clear that, once again, the Frodo scheme has a significantly more read and written bytes than every other scheme.

6.4 CONSIDERATIONS

Overall, all signature schemes seem to have more or less the same performance when only taking into account connection times. However, given that certificate generation varies a great deal depending on the algorithm, the ECDSA-P256 signature scheme seems to consistently perform better than every other algorithm, both with certificate generation times, certificate sizes, connection times and handshake data transfers.

Dilithium and qTeslaI (and the hybrid variants) have significantly larger certificate sizes, although generation times are comparable to those of ECDSA-P256. Furthermore, connection times seem to be similar, even when using the ECDSA-P256 hybrid variants. Handshake data increases significantly when using these algorithms, given the increased certificate sizes.

RSA variants don't seem to be the best options when using digital signatures, according to the metrics evaluated, since certificate generation is considerably slower than other schemes for the same security bits, and is all but impractical when trying to achieve higher security bits (to achieve 256 bits, RSA modulus would have to be around 15360 bits long). When comparing connections, RSA-3072 also doesn't provide any major advantage in connection times, and while it does have less data being exchanged during the handshake than post-quantum algorithms, it still has more than ECDSA-P256.

It seems to be the case that, if post-quantum signature algorithms are to be used on TLS, the impact is larger on the actual data being transferred during the TLS handshake, given the larger key sizes and signatures. However, if it is the case that these algorithms must be included on TLS communications, using a hybrid variant until there is a standard post-quantum algorithm which bears enough trust to actually replace classical algorithm seems to be an acceptable option, given the small amount of impact that hybrid signatures schemes seem to have both in connections times and data transferred, as well as certificate generation and size.

Classical and post-quantum key exchange schemes also seem to bear some differences in performance, although not as much in connection times, with the exception of the LWE based scheme Frodo (LWE schemes usually use larger keys) and SIKEp503. NewHope512 and ECDHE-P256 seem to have similar connection times, but NewHope, being a R-LWE scheme, again, involves a larger amount of bytes being exchanged during the TLS handshake. Its hybrid variant also exchanges more bytes than ECDHE-P256. ThreeBears, being based

on a variant of the M-LWE problem, involves a slightly shorter amount of bytes exchanged than NewHope, but still significantly higher than ECDHE-P256.

The impact of including a post-quantum key exchange scheme on TLS seems to bear similarities to that of including a digital signature, in which the negative impacts are clearly present in the amount of data being exchanged through the parties in the connection, and not so much on the actual connection times. While ECDHE-P256 clearly needs less data exchanged to establish a key exchange, if post-quantum security is to be included on TLS, hybrid key exchange schemes also seem to be an acceptable option, since the impact of using, for example, NewHope512 with ECDHE-P256 seems to be rather small when comparing to the impact of only including NewHope512.

7

CONCLUSIONS AND FUTURE WORK

Post-quantum cryptography is getting closer everyday to a possible standardization that might change the way most traffic on the internet gets secured. Initiatives like the NIST Post-Quantum Cryptography Standardization Process have already started the long process of determining which proposed algorithms are more suitable to eventually replace the classical schemes used today, some of which have been used for decades.

Because projects like Open Quantum Safe are currently accessible to anyone willing to experiment with post-quantum algorithms, and because they allow a somewhat simple integration with protocols like OpenSSL, companies and other entities can start protecting their traffic using quantum-safe alternatives, even if the trade-off is having to use hybrid schemes until a standard which bears enough confidence is released and deployed world-wide. The Open Quantum Safe Project, even though it still is described as a way to prototype post-quantum algorithms, makes these algorithms more accessible to be used and tested with a consistent API that can be used in very much the same way as the OpenSSL API has been used for years in multiple applications.

As was shown in chapter 6, the R-LWE quantum-safe algorithms that are currently in the "race" for standardization all bear similar results when compared to each other. The increased sizes for keys and signatures might have a significant impact on certain applications, but are at least compatible, as of now, with TLS, the most used protocol to safely transmit data over the internet, seemingly without any significant negative impact on the time each connection takes to establish. Furthermore, that conclusion had already been achieved in Google's post-quantum experiment [Bra16; Lan16], where it was deemed that any latency caused was mostly due to the increased message sizes. Albeit in a very small and somewhat controlled environment, the results obtained in this work seem to hold with the results from the NewHope post-quantum experiment held by Google, extended to more post-quantum algorithms, including digital signature schemes. If one of the algorithms tested ever becomes standardized by NIST, then it seems the biggest trade-off is in fact the increased sizes in certificates and message sizes in key exchange protocols, and thus if a certain entity decides to employ quantum-safe algorithms in its communications, the impact would most likely

be seen on the increased sizes in messages and certificates, and not so much on the time a connection takes.

Even when comparing quantum-safe algorithms outside of TLS, it seems that producing key pairs and signing messages is on the level of ECDSA schemes, given the fact that certificate generation times are approximately the same. Using RSA, however, seems to incur on a penalty in performance that only tends to increase as time progresses, since time for generating keys of higher bit security tends to increase exponentially.

The integration of post-quantum cryptography on other projects, like Python's **cryptography**, is also helpful by allowing the access to quantum-safe primitives in an even simpler and more accessible way, by abstracting from the low level C implementations and offering a simple, consistent interface that has also been used for years with classical algorithms. One possible future improvement on the integration accomplished in this thesis is the addition of other functionalities, like certificate generation, to **cryptography**, to complete the set of post-quantum functionalities offered by this package.

Throughout the literature, some authors, like Bos et al. [Bos+16] and Albrecht and Deo [AD17], raised some concerns about the extra algebraic structure present in schemes based on the R-LWE problem. Specifically, this problem, given the added algebraic complexity, might be more vulnerable to attacks that haven't yet been discovered that can exploit the added structure and render any cryptosystem based on R-LWE unsafe. It seems that there are some trade-offs to take notice when developing public key cryptography schemes. On one hand, having rich algebraic structures that have interesting properties to be explored when developing cryptosystems may lead to schemes that are efficient and practical to use, such as R-LWE when opposed to LWE. On the other, it might also make the scheme more prone to attacks, simply because the extra complexity introduce new ways of attack not present in simpler schemes.

It seems to be the case that R-LWE schemes (and to some extent, LWE schemes as well) are quite efficient, even when compared to other quantum-safe schemes. It may be the case that these problems are in fact hard, and that hardness might me strong enough to offer confidence in the schemes when it comes to protecting communications in the future. However, should that not be the case, it could be useful to have other schemes based on completely different problems (not based on lattices) that possess a much simpler algebraic structure. Schemes like Picnic [Cha+17] and Sphincs+ [Ber+17], for instance, could be a good alternative to LWE based schemes, should the latter be proven to be insecure somewhere in the future. Investigating schemes with simpler algebraic structures and their real-world viability might be a good follow-up to this work.

Overall, it seems that whether or not quantum computers powerful enough to run Shor's algorithm ever get built, there are some promising proposals based on the LWE problem and its variants, both for key exchange schemes and digital signatures, that aim to resist this

algorithm, and that, if maintaining the current increase in key and signature sizes, might still be viable to effectively protect communications between parties and thus maintain communications on the internet secure.

A

INTEGRATION ON PYTHON'S CRYPTOGRAPHY PACKAGE

A.1 INTRODUCTION

This guide aims at providing help on how to add new functionalities, such as new algorithms, to Python's **cryptography** package. **Cryptography** offers a lot of functionalities, such as symmetric and asymmetric encryption, digital signatures, key exchange schemes and a lot of operations using certificates both in PEM and DER formats. This guide will go through the process of integrating two of these classes of algorithms: digital signatures and key exchange schemes.

Without entering into too much detail, **cryptography** acts like a higher level interface for a general purpose library of cryptographic functions. The low-level implementations themselves are coded in a C library, which is then called by the interface. The "bridge" between the high-level interface and the C implementation of the algorithms used is the backend. The backend does most of the heavy-lifting, while abstracting from any implementation details that not necessary for using the package.

Originally, **cryptography** used to support the use of multiple backends, meaning that users could choose which algorithm implementations they preferred. However, in the most recent versions, support for multiple backends ceased, and thus the only backend currently supported is the OpenSSL backend. What this means is that every cryptographic algorithm used will be executed using the OpenSSL library.

Usually, adding new algorithms would require that the new code be integrated on OpenSSL, and then the interface to call it added to **cryptography**. However, it is not actually mandatory for the new code to belong to the OpenSSL library. In fact, **cryptography** allows for the inclusion of more C libraries when compiling. By default, it includes the *libssl* and *libcrypto* libraries, but other libraries, independent from OpenSSL and compiled from a different source, can be included as well. By adding new libraries, it's can avoid changing the OpenSSL source code altogether, provided the implementation of the new algorithm is coded elsewhere. However, if the objective is other than to just test algorithms locally, it could be the case that this approach would not be acceptable, since technically the OpenSSL

backend is no longer only using code from the OpenSSL libraries. Whether the C source code is implemented on OpenSSL or not, the process of integrating it on **cryptography** is fairly the same, and so it shouldn't be relevant where the actual source code comes from, as long as it is compiled into a library first.

The example used in this guide is the inclusion of a generic signature scheme implemented in the OpenQuantumSafe project. By building the source code, the necessary library will be created and added to */usr/local/lib*. The generic scheme can be seen as a wrapper for different algorithms, which can be called using the generic functions' parameters. The inclusion of a key exchange scheme is identical, except for the specific functions that need to be included in each case.

A.2 Including C library and new module

Assuming the new algorithm to introduce is in a separate library (as was mentioned above), there is a file that needs to be altered to include the extra library when compiling. The root of the **cryptography** package contains a folder called *src*. Every alteration needed to add a new algorithm is made inside the *src* folder.

To add a new library, one must modify the file *src/_cffi_src/build_openssl.py* by adding the library to the function *_get_openssl_libraries*, just like is shown in figure 33. The alteration is meant to work on Linux. If the objective is compiling on Windows, the alteration must be done in the code related to the Windows system.

```
def _get_openssl_libraries(platform):
    if os.environ.get("CRYPTOGRAPHY_SUPPRESS_LINK_FLAGS", None):
        return []
    # OpenSSL goes by a different library name on different operating systems.
    if platform == "win32" and compiler_type() == "msvc":
        windows_link_legacy_openssl = os.environ.get(
            "CRYPTOGRAPHY_WINDOWS_LINK_LEGACY_OPENSSL", None
        )
        if windows_link_legacy_openssl is None:
            # Link against the 1.1.0 names
            libs = ["libssl", "libcrypto"]
        else:
            # Link against the 1.0.2 and lower names
            libs = ["libeay32", "ssleay32"]
        return libs + ["advapi32", "crypt32", "gdi32", "user32", "ws2_32"]
    else:
        # darwin, linux, mingw all use this path
        # In some circumstances, the order in which these libs are
        # specified on the linker command-line is significant;
        # libssl must come before libcrypto
        # (https://marc.info/?l=openssl-users&m=135361825921871)
        return ["/usr/local/lib/oqs", "/usr/local/lib/ssl", "/usr/local/lib/crypto"]
```

Figure 33: Inclusion of *liboqs* library in **cryptography**

There is on more modification that must be done in this file, and it's related to the new module that will be created. Each module, such as the *dsa* module for example, needs to

be registered in this file. For the new example module, which will be called *oqs_sig*, it's necessary to add it to the list of modules to be compiled, just as is shown in 34.

```
ffi = build_ffi_for_binding(
    module_name="_openssl",
    module_prefix="_cffi_src.openssl.",
    modules=[
        # This goes first so we can define some cryptography-wide symbols.
        "cryptography",

        "aes",
        "asn1",
        "bignum",
        "bio",
        "cmac",
        "conf",
        "crypto",
        "ct",
        "dh",
        "oqs_kem",
        "oqs_sig",
        "dsa",
        "ec",
        "ecdh",
        "ecdsa",
        "engine",
```

Figure 34: Inclusion of *oqs_sig* module in **cryptography**.

After applying these modifications, no further action is required in this file.

A.3 Including the C functions

After modifying *build_openssl.py*, the next step to do with including the C library is creating a file that will register every C function from the new library that will be included on **cryptography**. The file must be created in the *src/_cffi_src/openssl* directory, which contains a file for most algorithms and other functionalities implemented in OpenSSL. For example, there is a file for DSA, called *dsa.py*, and the same goes for Diffie-Hellman, RSA and all the elliptic curve variants. Each file registers the C structures, functions and other variables, as well as the include files where those elements are declared.

Following the naming of the already present files, it makes sense to create a new file named *oqs_sig.py*. Regardless of what the module is called, it must match the name of the module added in *build_openssl.py*.

Inside the file, there is a structure that must be followed for declarations. The easiest way to understand how the file should be structured is by following the example of an already present algorithm, such as DSA. Nevertheless, figure 35 shows how the declarations should be done in order for the integration to be properly accomplished.

```
from __future__ import absolute_import, division, print_function

INCLUDES = """
#include <oqs/oqs.h>
"""

TYPES = """
typedef ... OQS_SIG;
typedef ... OQS_STATUS;
"""

FUNCTIONS = """
OQS_SIG *OQS_SIG_new(const char *);
int OQS_SIG_keypair(const OQS_SIG *, uint8_t *, uint8_t *);
int OQS_SIG_sign(const OQS_SIG *, uint8_t *, size_t *, const uint8_t *, size_t , const uint8_t *);
int OQS_SIG_verify(const OQS_SIG *, const uint8_t *, size_t , const uint8_t *, size_t , const uint8_t *);
int OQS_SIG_DILITHIUM_verify(const uint8_t *, size_t , const uint8_t *, size_t , const uint8_t *);
size_t OQS_SIG_pkey_length(OQS_SIG *);
size_t OQS_SIG_pub_length(OQS_SIG *);
size_t OQS_SIG_signature_length(OQS_SIG *);
char* OQS_SIG_get_algorithm(OQS_SIG *);

void OQS_MEM_secure_free(void *, size_t );
void OQS_MEM_insecure_free(void *);
void OQS_SIG_free(OQS_SIG *);
void OQS_randombytes(uint8_t *, size_t );
int memcmp(const void *, const void *, size_t);
"""

CUSTOMIZATIONS = """
"""
```

Figure 35: Variable and function declarations on the *oqs_sig* module.

Most C types used in OQS are already declared in other files, such as the *uint8_t* type, which means that the only additional types needed for this declaration are the OQS_SIG struct and the OQS_STATUS variable, which is an enum type (using only integers). These types need to be declared because most added functions involve the use of a OQS_SIG struct. The functions themselves should be declared as is seen in 35, where parameters do not have names, only their types. Every function declared will later be usable in Python directly using the CFFI interface [RF18] (the modifications so far are all related to adding the C library to the CFFI interface that is implemented on **cryptography**). So, even though these steps add the new functionalities to CFFI, there is no need to actually delve deep into how the whole interface works.

With the modifications made to *build_openssl.py* and the new *oqs_sig.py* files, it is actually possible to use the new functions in **cryptography** already, by using the CFFI interface directly. That involves including the binding module and then creating a new binding object that will include a variable called *lib*, which contains every C function registered under the *src/_cffi_src/openssl* directory. However, this approach, while sufficient to use the new algorithms from Python, doesn't provide a high-level interface to the programmer, since it still requires that the necessary variables be allocated using, for example, *malloc*, the same way it would be done directly in C, while possibly being even more confusing, since variables casts, for example, are accomplished in a different way when using CFFI.

A.4 CREATING THE INTERFACE FOR OQS_SIG

At this point, no other modifications under *src/_cffi_src* are needed, the rest of them taking place under *src/cryptography*.

When a programmer needs to use the DSA module from **cryptography**, it needs to import the correcct module from the package. Usually, it will look like this:

from cryptography.hazmat.primitives.asymmetric import dsa

The path shown in the line above corresponds to the folder path *src/cryptography/hazmat/primitives/asymmetric*.

The *dsa.py* file includes, essentially, definitions of classes related to the DSA algorithm, such as DSAParameters, DSAPrivateKey and DSAPublicKey, among others. These are, however, merely definitions, not actually the real implementations of those classes, working instead as interfaces for the real implementations. Besides the class definitions, *dsa.py* also has several functions that correspond to the functions the programmer can call when importing the DSA module. For example, the programmer can call the *generate_private_key* function from the dsa module directly, which means that this function is defined in this file.

So, in order to add the module *oqs_sig*, it is necessary to create a file *oqs_sig.py* implementing a similar interface and necessary functions to correctly implement the algorithm. For the most part, the structure in *dsa.py* can be followed without too many alterations, but it is worth to keep in mind, however, that the classes to create for each algorithm depend on the algorithm itself. It doesn't make sense to create a whole OQS_SIGParameters class if the parameters are fixed in the C implementation, for example.

To follow the *dsa.py* example, the only really important function needed in this file is the *generate_private_key*, which, in this concrete example, receives as parameters the algorithm name and the backend itself (as opposed to the DSA variant which receives the key size and backend). The key size is fixed for each algorithm passed as parameter to the *oqs_sig* module (the liboqs C library implements algorithms with fixed parameters), which means it is not necessary to be passed as a parameter in this case.

Following the most basic example possible, the *oqs_sig.py* file created must contain:

- Class definition for PrivateKey (name could be OQS_SIGPrivateKey)
- Class definition for PublicKey (name could be OQS_SIGPublicKey)
- Function to generate a private key (can be called just *generate_private_key*)

Figure 36 shows the interface for OQS_SIGPublicKey and the definition of the *generate_private_key* function, the function that will be called directly from the *oqs_sig* module.

```
@six.add_metaclass(abc.ABCMeta)
class OQSPublicKey(object):
    @abc.abstractproperty
    def key_size(self):
        """
        The length of the public key.
        """

    @abc.abstractmethod
    def verify(self, signature, data, algorithm):
        """
        Verifies the signature of the data.
        """

def generate_private_key(algorithm, backend):
    return backend.generate_oqs_private_key(algorithm)
```

Figure 36: Example showing the interface defined in *oqs_sig.py*.

After creating the file, the next place to make modifications will be the backend itself.

A.5 MODIFYING THE BACKEND CLASS

When using the *generate_private_key* function, one of the parameters received is the backend where the algorithms are implemented. Since the only supported backend is OpenSSL, the user usually passes the backend object as the return value from the function *default_backend*.

Internally, the only thing the *generate_private_key* function does is call the function for generating the private key of a specific algorithm defined in the backend. Because the *oqs_sig* algorithm is new, every specific function the backend must implement for this specific algorithm will need to be created from scratch. This means that, for example, the backend function *generate_oqs_sig_private_key* (which can be seen being called in figure 36) needs to be implemented on the backend class.

The path where every modification needs to be made from now on is *src/cryptography/hazmat/backends*. First of all, the file *interfaces.py* needs to be modified, by adding a new interface for the backend class. Every algorithm specific function implemented by the backend is defined here, similarly to what happens with *oqs_sig.py*, mentioned in A.4. In this case, the only function implemented is the *generate_oqs_sig_private_key*. The added interface must look like what is shown in figure 37.

```
@six.add_metaclass(abc.ABCMeta)
class OQSBackend(object):
    @abc.abstractmethod
    def generate_oqs_private_key(self, algorithm):
        """
        Return a OQSPrivateKey object, which in turn must
        """
```

Figure 37: Example showing the interface for OQS_SIGBackend.

It's important to note that the backend class will implement this new interface as well, so the function naming, as well as the parameters in the actual implementation, should match the interface.

After adding the new interface, the next step is to alter the backend class itself, to include the new function to generate a private key for the new algorithm. The file containing the implementation of the class is located under the *backends/openssl*, and is called *backend.py*.

There are some lines that need to be edited or added in this file. First and foremost, the new interface needs to be imported in the beginning of the file, following the example from the other interfaces. The new interface must also be registered, following the example from other interfaces as well (figure 38).

```
@utils.register_interface(CipherBackend)
@utils.register_interface(CMACBackend)
@utils.register_interface(DERSerializationBackend)
@utils.register_interface(DHBackend)
@utils.register_interface(DSABackend)
@utils.register_interface(OQSBackend)
@utils.register_interface(OQS_KEMBackend)
@utils.register_interface(EllipticCurveBackend)
@utils.register_interface(HashBackend)
@utils.register_interface(HMACBackend)
@utils.register_interface(PBKDF2HMACBackend)
@utils.register_interface(RSABackend)
@utils.register_interface(PEMSerializationBackend)
@utils.register_interface(X509Backend)
@utils.register_interface_if(
    binding.Binding().lib.Cryptography_HAS_SCRYPT, ScryptBackend
)
```

Figure 38: Figure showing the interface registration for various interfaces in the *backend.py* file.

After these two additions, the actual function must be implemented in the class itself. The function *generate_oqs_sig_private_key*, at this point, only receives the algorithm string as a parameter. The whole code that makes use of the CFFI interface to make C function calls and allocate variables will be implemented here. The specific implementation will differ between algorithms, but the logic itself is essentially the same. Since the objective

is to return a PrivateKey object (which was declared before in *oqs_sig.py*), the operations needed will involve generating a private key from the C implementation and then creating a PrivateKey object that contains the private key within itself. Figure 39 shows the example for the *oqs_sig* generic algorithm. The logic is similar to what one would expect when coding in C, allocating variables using *malloc* and then initializing a PrivateKey object with these variables.

```
def generate_oqs_private_key(self, algorithm):
    sig = self._lib.OQS_SIG_new(algorithm)
    self.openssl_assert(sig != self._ffi.NULL)
    sig = self._ffi.gc(sig, self._lib.OQS_SIG_free)
    pkey = self._ffi.cast("uint8_t *", self._lib.OPENSSL_malloc(self._lib.OQS_SIG_pkey_length(sig)))
    pub = self._ffi.cast("uint8_t *", self._lib.OPENSSL_malloc(self._lib.OQS_SIG_pub_length(sig)))
    pkey = self._ffi.gc(pkey, self._lib.OPENSSL_free)
    pub = self._ffi.gc(pub, self._lib.OPENSSL_free)
    #do not forget to free memory after use to prevent leaking
    ret = self._lib.OQS_SIG_keypair(sig, pub, pkey)
    self.openssl_assert(ret == 0)
    return _OQSPrivateKey(self, pkey, pub, sig)
```

Figure 39: Implementation of *generate_oqs_sig_private_key* in the backend class.

The final step to complete the basic integration of the new algorithm is implementing the classes defined before in the interface.

A.6 Defining the OQS_SIG specific classes

The *generate_oqs_sig_private_key* return an object that represents the private key used to create signatures. This object's class must be defined somewhere, which in this case is a file on the same directory of *backend.py*, which will also be named *oqs_sig.py*. To check which modules are necessary to import, the easiest way is, again, to check the example of the DSA implementation in *dsa.py* on the same directory. The most important module to include is the module corresponding to the algorithm to be implemented, which was already created (it's the file *oqs_sig.py* created in *src/cryptography/hazmat/primitives/asymmetric*).

The next step is to implement every class previously defined in the interface, which in this case is just the OQS_SIGPrivateKey and OQS_SIGPublicKey. The implementation must include the functions as defined in the interface. Figures 40 and 41 show, respectively, the implementations of OQS_SIGPrivateKey and OQS_SIGPublicKey. The *_oqs_sig_sign* and *_oqs_sig_verify* functions must also be defined in the same file.

```
@utils.register_interface(oqs_sig.OQSPrivateKey)
class _OQSPrivateKey(object):
    def __init__(self, backend, pkey, pub, sig):
        self._backend = backend
        self._pkey = pkey
        self._pub = pub
        self._sig = sig

    key_size = utils.read_only_property("_key_size")

    def public_key(self):
        return _OQSPublicKey(self._backend, self._pub, self._sig)

    def sign(self, data, algorithm):
        data, algorithm = _calculate_digest_and_algorithm(
            self._backend, data, algorithm
        )
        return _oqs_sig_sign(self._backend, self, data, self._sig)
```

Figure 40: Implementation of the OQS_SIGPrivateKey class.

```
@utils.register_interface(oqs_sig.OQSPublicKey)
class _OQSPublicKey(object):
    def __init__(self, backend, pub, sig):
        self._backend = backend
        self._pub = pub
        self._sig = sig

    key_size = utils.read_only_property("_key_size")

    def verify(self, signature, data, algorithm):
        data, algorithm = _calculate_digest_and_algorithm(
            self._backend, data, algorithm
        )
        return _oqs_sig_verify(self._backend, self, signature, data, self._sig)
```

Figure 41: Implementation of the OQS_SIGPublicKey class.

When using the DSA module on **cryptography**, the user first generates a DSAPrivateKey, which is then used to generate the corresponding DSAPublicKey. The private key generated is then used to sign a message by invoking the *sign* function from the DSAPrivateKey object. The verification, analogously, is accomplished by using the *verify* function associated with the DSAPublicKey object. To follow this strucuture, the new algorithm's keys must also implement the methods in the same way. This will give the programmer a consistent interface that is more or less independent from the algorithm used. Regardless of the algorithm in question, the programmer always generates a private key, then a public key, and then signs the message with the private key and verifies with the public.

Depending on the algorithm implemented, the sign and verify functions defined in these classes will naturally differ. However, no matter the algorithm, it is in these functions that the

C code will be called to respectively sign and verify the message, similarly to what was done with the key generation in the backend class. The *sign* function (which calls *_oqs_sig_sign*) must return a signature and the *verify* function (which calls *_oqs_sig_verify*) must receive the signature and verify its validity. The actual type of the signature isn't really important, as long as the data is treated properly in both functions.

To finalize, it's important to make a final modification in the *backend.py*, which is importing the class implementations from *oqs_sig.py*. Figure 42 shows how to add this modification.

```
from cryptography.hazmat.backends.openssl.oqs_sig import (
    _OQSPrivateKey, _OQSPublicKey
)
```

Figure 42: Example of importing OQS_SIG classes in the *backend.py* file.

A.7 SUMMARY

To sum up, the files that need to be altered (with the full path) are:

- *src/_cffi_src/build_openssl.py*
- *src/_cffi_src/openssl/new_algorithm.py*
- *src/cryptography/hazmat/primitives/asymmetric/new_algorithm.py*
- *src/cryptography/backends/interfaces.py*
- *src/cryptography/backends/openssl/backend.py*
- *src/cryptography/backends/openssl/new_algorithm.py*

When all modifications are in place, the final step is to install the package. To install, it's necessary to run the following command in the root of the **cryptography** package's source code (the directory where the folder *src* is located):

sudo python setup.py install

To install the package successfully, besides having Python installed, it is also needed to have the CFFI package installed. To check if it's installed, one can just type *pip freeze* in the terminal. If it's not, it can be installed by typing *pip install cffi*.

Installing the **cryptography** package with the command above results in the same effect as installing it with *pip install* ***cryptography***, so the package should appear when typing *pip freeze* no matter how it was installed.

When successfully installed, the package will now allow the programmer to sign and verify messages in the same way as with the DSA algorithm, by first importing the *oqs_sig*

module, generating a private key and then using the key to generate a public key. The private key will then be used to sign an arbitrary message and the public key to verify it.

Even though the examples shown were to integrate a very basic generic signature algorithm, the process to include different kinds of algorithms, like KEMs, is very similar.

A.8 USAGE EXAMPLE

After the integration is completed successfully, one can quickly test if it's working properly by creating a simple Python script to sign and verify a message, for example.

To use any OQS signatures, the imports that need to be made are identical to those of the DSA algorithm, sufficing it to replace the dsa module that is imported from *cryptography.hazmat.primitives.asymmetric* with *oqs_sig*. Other imports needed are, as usual, the default_backend, hashes and InvalidSignature modules.

The only difference worth noting from using an OQS algorithm, like Dilithium for instance, and using DSA, is the parameters passed to the *generate_private_key* function. When using OQS, the only parameter needed, other than the backend, is the name of the algorithm to be used, contrarily to what happens with DSA, which receives the key size as a parameter. This difference exists because the *oqs_sig* module does not implement a specific algorithm, but instead a number of different algorithms. Parameters are fixed in the C implementation of the liboqs library. For different parameters, like key sizes, there needs to be a different algorithm name. For instance, qTesla_I and qTesla_IIIsize are two distinct algorithms that can be used in the *oqs_sig* module, yet they represent the same scheme, qTesla, only with different parameters.

Figure 43 shows a very simple example of how the *oqs_sig* module can be used to sign a string using **cryptography**.

```
from __future__ import print_function
from cryptography.hazmat.primitives.asymmetric import oqs_sig
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.exceptions import InvalidSignature

import sys


def main():

        if (len(sys.argv) < 2):
                return
        try:

                private_key = oqs_sig.generate_private_key(sys.argv[1], default_backend())

                public_key = private_key.public_key()

                data = "This is the message to be signed."

                signature = private_key.sign(data, hashes.SHA256())

                public_key.verify(signature, data, hashes.SHA256())

        except InvalidSignature:
                print("Message verification failed!")
                return

        print("Verified successfully!")

if __name__=="__main__":
        main()
```

Figure 43: Signature example using the *oqs_sig* module.

The variable *sys.argv[1]* is the name of the algorithm to be used, which is not case sensitive.

The process is quite similar when using the KEM algorithms in OQS. One must first import the *oqs_kem* module (integrated in the same way as the *oqs_sig* module) and other necessary modules. Right after, the private and public keys can be generated in the exact same way as with any signature algorithm (the keys belong to Alice). The *oqs_kem* module has a new function that does not exist in the DH-like functions already present on **cryptography**, which is called *generate_ciphertext*, and which objective is to generate the message that Bob will send to Alice. This message will contain Bob's public key, to achieve an approximate shared key, and some additional information that Bob needs to send to achieve an exact shared key between the two entities.

The function only needs to return the ciphertext necessary to achieve a shared secret, but for purposes of confirming that the algorithm is working properly, the shared key produced by Bob is also outputted.

The final step is invoking the exchange method from the private key giving the ciphertext as a parameter to output the shared key that Alice will end up with.

Figure 44 shows an example of a key agreement using *oqs_kem*.

```
from cryptography.hazmat.primitives.asymmetric import oqs_kem
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.bindings.openssl import binding
import sys


def main():

        argc = len(sys.argv)

        if (argc < 2):
                return

        try:
                private_key = oqs_kem.generate_private_key(sys.argv[1], default_backend())

                public_key = private_key.public_key()

                ciphertext, shared_b = oqs_kem.generate_ciphertext(default_backend(), public_key)

                shared_a = private_key.exchange(ciphertext)

                p = binding.Binding()

                if(p.lib.memcmp(shared_a, shared_b, 32) != 0):
                        print("Shared keys do not match.")
                        return

        except Exception, e:
                print("Shared keys do not match.")
                print("EXCEPTION: " + str(e))
                return

        print("Key exchange successful!")
```

Figure 44: Key agreement example using the *oqs_kem* module.

The last lines where there is a *binding.Binding()* exists only to compare the shared secrets for testing purposes, and do not need to be present for the algorithm to work.

BIBLIOGRAPHY

[AD17] Martin R. Albrecht and Amit Deo. *Large Modulus Ring-LWE ≥ Module-LWE*. Cryptology ePrint Archive, Report 2017/612. https://eprint.iacr.org/2017/612. 2017.

[AD97] Miklós Ajtai and Cynthia Dwork. "A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence." In: Jan. 1997, pp. 284–293.

[Akl+16a] Sedat Akleylek, Nina Bindel, Johannes Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. *An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation*. Cryptology ePrint Archive, Report 2016/030. https://eprint.iacr.org/2016/030. 2016.

[Akl+16b] Sedat Akleylek, Nina Bindel, Johannes Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. *Cryptology ePrint Archive*. 2016. URL: https://eprint.iacr.org/2016/030.

[Alb+12] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. *On the Complexity of the BKW Algorithm on LWE*. Cryptology ePrint Archive, Report 2012/636. https://eprint.iacr.org/2012/636. 2012.

[Alk+15] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. *Post-quantum key exchange - a new hope*. Cryptology ePrint Archive, Report 2015/1092. https://eprint.iacr.org/2015/1092. 2015.

[Alk+16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. *NewHope without reconciliation*. Cryptology ePrint Archive, Report 2016/1157. https://eprint.iacr.org/2016/1157. 2016.

[Alk+19] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Patrick Longa, and Jefferson E. Ricardini. *The Lattice-Based Digital Signature Scheme qTESLA*. Cryptology ePrint Archive, Report 2019/085. https://eprint.iacr.org/2019/085. 2019.

[App+09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. "Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems". In: *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '09. Santa Barbara, CA: Springer-Verlag, 2009, pp. 595–618. ISBN: 978-3-642-03355-1. DOI: 10.1007/978-3-642-03356-8_35. URL: https://doi.org/10.1007/978-3-642-03356-8_35.

[Bar+16] Paulo S. L. M. Barreto, Patrick Longa, Michael Naehrig, Jefferson E. Ricardini, and Gustavo Zanon. *Sharper Ring-LWE Signatures*. Cryptology ePrint Archive, Report 2016/1026. https://eprint.iacr.org/2016/1026. 2016.

[Ber+17] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukasz Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, and Peter Schwabe. *SPHINCS$^+$: Submission to the NIST post-quantum project*. Submission to the NIST Post-Quantum Cryptography Standardization Project. https://cryptojedi.org/papers/#sphincsnist. 2017.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. https://eprint.iacr.org/2011/277. 2011.

[Bos+14] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. *Post-quantum key exchange for the TLS protocol from the ring learning with errors problem*. Cryptology ePrint Archive, Report 2014/599. https://eprint.iacr.org/2014/599. 2014.

[Bos+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. *Frodo: Take off the ring! Practical, Quantum-Secure Key Exchange from LWE*. Cryptology ePrint Archive, Report 2016/659. https://eprint.iacr.org/2016/659. 2016.

[Bos+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM*. Cryptology ePrint Archive, Report 2017/634. https://eprint.iacr.org/2017/634. 2017.

[Bos+19] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS – Kyber: Algorithm Specifications and Supporting Documentation (version 2.0)*. https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf. 2019.

[Bra16] Matt Braithwaite. *Experimenting with Post-Quantum Cryptography*. 2016. URL: https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[BWB18] Denis Butin, Julian Wälde, and Johannes Buchmann. "Post-quantum authentication in OpenSSL with hash-based signatures". In: *2017 10th International Conference on Mobile Computing and Ubiquitous Network, ICMU 2017* 2018-Janua (2018), pp. 1–6. DOI: 10.23919/ICMU.2017.8330093.

[Cha+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. *Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives*. Cryptology ePrint Archive, Report 2017/279. https://eprint.iacr.org/2017/279. 2017.

[Cho17] Arjun Chopra. "GLYPH: A New Instantiation of the GLP Digital Signature Scheme". In: (2017), pp. 1–14. URL: https://eprint.iacr.org/2017/766.pdf.

[Chu17] Gu Chunsheng. *Integer Version of Ring-LWE and its Applications*. Cryptology ePrint Archive, Report 2017/641. https://eprint.iacr.org/2017/641. 2017.

[DDW18] Xiaoyang Dong, Bingyou Dong, and Xiaoyun Wang. *Quantum Attacks on Some Feistel Block Ciphers*. Cryptology ePrint Archive, Report 2018/504. https://eprint.iacr.org/2018/504. 2018.

[DTV15] Alexandre Duc, Florian Tramèr, and Serge Vaudenay. *Better Algorithms for LWE and LWR*. Cryptology ePrint Archive, Report 2015/056. https://eprint.iacr.org/2015/056. 2015.

[Duc+13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. *Lattice Signatures and Bimodal Gaussians*. Cryptology ePrint Archive, Report 2013/383. https://eprint.iacr.org/2013/383. 2013.

[Duc+17a] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium : A Lattice-Based Digital Signature Scheme*. 2017.

[Duc+17b] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. *CRYSTALS – Dilithium: Digital Signatures from Module Lattices*. Cryptology ePrint Archive, Report 2017/633. https://eprint.iacr.org/2017/633. 2017.

[DW17] Xiaoyang Dong and Xiaoyun Wang. *Quantum Key-recovery Attack on Feistel Structures*. Cryptology ePrint Archive, Report 2017/1199. https://eprint.iacr.org/2017/1199. 2017.

[DXL12] Jintai Ding, Xiang Xie, and Xiaodong Lin. *A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem*. Cryptology ePrint Archive, Report 2012/688. https://eprint.iacr.org/2012/688. 2012.

[Esp+17] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. *Side-Channel Attacks on BLISS Lattice-Based Signatures – Exploiting Branch Tracing Against strongSwan and Electromagnetic Emanations in Microcontrollers*. Cryptology ePrint Archive, Report 2017/505. https://eprint.iacr.org/2017/505. 2017.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 537–554. ISBN: 978-3-540-48405-9.

[Fot15] Jim Foti. "FIPS PUB 202 SHA3 Standard: PermutationBased Hash and ExtendableOutput Functions CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY". In: August (2015). ISSN: 00778923. DOI: 10.6028/NIST.FIPS.202.

[FS87] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Proceedings on Advances in cryptology—CRYPTO '86*. Santa Barbara, California, USA: Springer-Verlag, 1987, pp. 186–194. ISBN: 0-387-18047-8. URL: http://dl.acm.org/citation.cfm?id=36664.36676.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. "Public-Key Cryptosystems from Lattice Reduction Problems". In: *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '97. London, UK, UK: Springer-Verlag, 1997, pp. 112–131. ISBN: 3-540-63384-7. URL: http://dl.acm.org/citation.cfm?id=646762.706185.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 530–547. ISBN: 978-3-642-33027-8.

[GPV07] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. *Trapdoors for Hard Lattices and New Cryptographic Constructions*. Cryptology ePrint Archive, Report 2007/432. https://eprint.iacr.org/2007/432. 2007.

[Gro96] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 212–219. ISBN: 0-89791-785-5. DOI: 10.1145/237814.237866. URL: http://doi.acm.org/10.1145/237814.237866.

[Ham19] Mike Hamburg. *Three Bears*. https://www.shiftleft.org/papers/threebears/nist-round2.pdf. 2019.

[HPS01] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. "NSS: An NTRU Lattice-Based Signature Scheme". In: *Advances in Cryptology — EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 211–228. ISBN: 978-3-540-44987-4.

[JN16] Piotr Jurkiewicz and Marcin Niemiec. "Implementation of a New Cipher in OpenSSL Environment the Case of INDECT Block Cipher". In: *International Journal of Computer and Communication Engineering* 5.1 (2016), pp. 41–49. ISSN: 20103743. DOI: 10.17706/IJCCE.2016.5.1.41-49. URL: http://www.ijcce.org/index.php?m=content%7B%5C&%7Dc=index%7B%5C&%7Da=show%7B%5C&%7Dcatid=56%7B%5C&%7Did=508.

[Jur12] Piotr Jurkiewicz. "Integration of Indect Block Cipher into the OpenSSL library". In: December (2012).

[Käs12] Emilia Käsper. "Fast elliptic curve cryptography in OpenSSL". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7126 LNCS (2012), pp. 27–39. ISSN: 03029743. DOI: 10.1007/978-3-642-29889-9_4.

[KV09] Jonathan Katz and Vinod Vaikuntanathan. "Smooth Projective Hashing and Password-Based Authenticated Key Exchange from Lattices". In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by Mitsuru Matsui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 636–652. ISBN: 978-3-642-10366-7.

[KV10] Jonathan Katz and Vinod Vaikuntanathan. *Round-Optimal Password-Based Authenticated Key Exchange*. Cryptology ePrint Archive, Report 2010/368. https://eprint.iacr.org/2010/368. 2010.

[Lan16] Adam Langley. *CECPQ1 results*. 2016. URL: https://www.imperialviolet.org/2016/11/28/cecpq1.html.

[LM08] Vadim Lyubashevsky and Daniele Micciancio. "Asymptotically Efficient Lattice-based Digital Signatures". In: *Proceedings of the 5th Conference on Theory of Cryptography*. TCC'08. New York, USA: Springer-Verlag, 2008, pp. 37–54. ISBN: 3-540-78523-X, 978-3-540-78523-1. URL: http://dl.acm.org/citation.cfm?id=1802614.1802619.

[LP10] Richard Lindner and Chris Peikert. *Better Key Sizes (and Attacks) for LWE-Based Encryption*. Cryptology ePrint Archive, Report 2010/613. https://eprint.iacr.org/2010/613. 2010.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13190-5.

[Lyu09] Vadim Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures". In: *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances*

in Cryptology. ASIACRYPT '09. Tokyo, Japan: Springer-Verlag, 2009, pp. 598–616. ISBN: 978-3-642-10365-0. DOI: 10.1007/978-3-642-10366-7_35. URL: http://dx.doi.org/10.1007/978-3-642-10366-7_35.

[Lyu11] Vadim Lyubashevsky. *Lattice Signatures Without Trapdoors*. Cryptology ePrint Archive, Report 2011/537. https://eprint.iacr.org/2011/537. 2011.

[MP11] Daniele Micciancio and Chris Peikert. *Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller*. Cryptology ePrint Archive, Report 2011/501. https://eprint.iacr.org/2011/501. 2011.

[Pei08] Chris Peikert. *Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem*. Cryptology ePrint Archive, Report 2008/481. https://eprint.iacr.org/2008/481. 2008.

[Pei14] Chris Peikert. "Lattice Cryptography for the Internet". In: *Post-Quantum Cryptography*. Ed. by Michele Mosca. Cham: Springer International Publishing, 2014, pp. 197–219. ISBN: 978-3-319-11659-4.

[Reg05] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*. STOC '05. Baltimore, MD, USA: ACM, 2005, pp. 84–93. ISBN: 1-58113-960-8. DOI: 10.1145/1060590.1060603. URL: http://doi.acm.org/10.1145/1060590.1060603.

[RF18] Armin Rigo and Maciej Fijalkowski. https://cffi.readthedocs.io/en/latest/. 2018.

[Sho94] P.W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124–134. ISSN: 0272-5428. DOI: 10.1109/SFCS.1994.365700. arXiv: 9605043 [quant-ph]. URL: http://ieeexplore.ieee.org/document/365700/.

[sJA19] Hwajeong soe, Amir Jalali, and Reza Azarderakhsh. *SIKE Round 2 Speed Record on ARM Cortex-M4*. Cryptology ePrint Archive, Report 2019/535. https://eprint.iacr.org/2019/535. 2019.

[SM16] Douglas Stebila and Michele Mosca. *Post-Quantum Key Exchange for the Internet and the Open Quantum Safe Project*. Cryptology ePrint Archive, Report 2016/1017. https://eprint.iacr.org/2016/1017. 2016.

[SS11] Damien Stehlé and Ron Steinfeld. "Making NTRU As Secure As Worst-case Problems over Ideal Lattices". In: *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT'11. Tallinn, Estonia: Springer-Verlag, 2011, pp. 27–47. ISBN: 978-3-642-20464-7. URL: http://dl.acm.org/citation.cfm?id=2008684.2008690.

[ST16] National Institute of Standards and Technology. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 2016.

[ST17] National Institute of Standards and Technology. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 2017.